池田尚史
[日] 藤仓和明 著
井上史彰

严圣逸 译

TURING 图灵程序设计丛书

提高项目质量、加快开发速度、降低运维成本

高效团队开发
工具与方法



版本管理 · Git · GitHub · 缺陷管理 · 持续集成 · Jenkins
Build工具 · 测试代码 · 部署 · 持续交付 · Vagrant
Chef · serverspec · Capistrano · 回归测试 · Selenium

中国工信出版集团 人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

池田尚史

DeNA软件开发工程师。曾做过IT顾问、程序员，从事过软件包开发、Web服务开发。Java的Web应用框架Play Framework 1的提交者。负责本书第1章~第5章，其中第2章的案例分析都是基于自身的实际经验编写的。

Twitter @ikeike443

藤仓和明

想能（SHANON）基础设施工程师。负责公司内部基础设施及服务环境的安全保障，致力于推动应用部署的自动化，并基于这方面丰富的实践经验，完成了本书第6章。喜欢OpenVZ、LXC等容器型虚拟化技术。

Twitter @fujya

井上史彰

想能（SHANON）软件工程师、QA工程师，现为想能信息科技（上海）有限公司总经理。开发经验丰富，致力于推动高效的自动化测试。负责本书第7章。

E-mail fu.inoue@gmail.com

严圣逸

毕业于上海交通大学。8年软件开发经验，期间赴日本工作。现就职于想能信息科技（上海）有限公司，从事基于云平台的客户关系管理及各类营销自动化系统的开发，侧重于对持续集成、自动化部署、自动化测试以及相关的开源工具的研究。本书所介绍的即是译者日常工作中所应用的开发流程以及工具。

封面设计 | 83720326@qq.com

TURING 图灵程序设计丛书

高效团队开发
工具与方法

池田尚史
[日] 藤仓和明 著
井上史彰

严圣逸 译



人民邮电出版社
北京

图书在版编目（CIP）数据

高效团队开发：工具与方法 /（日）池田尚史，（日）藤仓和明，（日）井上史彰著；严圣逸译. -- 北京：人民邮电出版社，2015.6
（图灵程序设计丛书）
ISBN 978-7-115-29594-1

Ⅰ. ①高… Ⅱ. ①池… ②藤… ③井… ④严… Ⅲ. ①程序设计 Ⅳ. ①TP311.1

中国版本图书馆CIP数据核字（2015）第109228号

内 容 提 要

本书以团队开发中所必需的工具的导入方法和使用方法为核心，对团队开发的整体结构进行概括性的说明。内容涉及团队开发中发生的问题、版本管理系统、缺陷管理系统、持续集成、持续交付以及回归测试，并且对“为什么用那个工具”“为什么要这样使用”等开发现场常有的问题进行举例说明。

本书适合所有想要系统性地学习团队开发工具的人阅读。

◆ 著　　　　[日]池田尚史　藤仓和明　井上史彰
　译　　　　严圣逸
　责任编辑　乐　馨
　执行编辑　杜晓静
　责任印制　杨林杰

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
　邮编　100164　　电子邮件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京　　　　　　印刷

◆ 开本：880×1230　1/32
　印张：10
　字数：298千字　　　　　　2015年6月第1版
　印数：1-4 000册　　　　　2015年6月北京第1次印刷

著作权合同登记号　图字：01-2014-7807号

定价：49.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京崇工商广字第0021号

致中文版的读者

感谢您购买了《高效团队开发：工具与方法》。同样要感谢已经决定购买本书的读者。还在犹豫是否购买本书的读者，如果您看了这篇序后决定购买，那将没有比这更令人高兴的事情了。

本书是由我、想能（SHANON）时期的同事藤仓和明先生与井上史彰先生共同写作的。这次是受到想能上海分公司总经理井上先生的委托来写的这篇序。

其实我并没有去过中国大陆。因此，从真正意义上来说，我并不知道中国软件行业的真实情况。当然，像阿里巴巴和腾讯这样的大公司还是知道的，但我没有在中国工作的经历。

中国的软件工程师极为优秀。我所在的公司就有很多非常优秀的中国工程师。OSS 社区也经常能看到中国的工程师，他们都十分令人敬佩。

中国工程师无疑是非常出众的，但中国的软件开发环境是怎样的呢？缺陷管理和分布式版本管理的运用，测试代码的覆盖和 CI 的配备，部署的自动化等机制的组合应用，这些我听说都还刚刚起步。

您所在的开发现场又是怎样的呢？

如果上述情形并未出现在您的开发现场中，那么非常抱歉，您不需要这本书。请将这本书放回书架，回去继续工作。如果存在上述情形，那么本书将对您有所帮助。

在日本，能够构建本书中所写的高效开发环境的公司和无法构建这样的环境的公司之间有着很大的差距。究其原因，其一是完备的环境有助于提高开发效率，能够迅速地发布优秀的产品或服务；其二是因为工程师注重团队开发环境是否完善，开发环境完善的公司能够吸引到优秀的工程师，而优秀的工程师越多开发效率自然就越高。这样一来，公司之间的差距就越来越大，这就是日本的真实情况。

同样的情况在中国也会发生，或许可能早就已经在发生了。

团队开发环境的完善就像“减肥”一样。明明知道只要去做就会有

效果、有益处，但却迟迟没有付之于行动。往往会以太忙了、有其他优先度更高的工作为由来应付过去。

本书第 2 章讲述了如果怠慢这个“减肥”会变成什么情形。那是我过去的真实体验，为了避免重复那样的经历，只要是能够提高团队开发效率的事情，我都会去尝试、实践，而本书就是这些尝试和实践的结晶。

请大家务必阅读、学习本书，避免陷入第 2 章中描述的悲惨境地。已经陷入上述境地的各位，更应该阅读本书，以便能够从上述境地中解脱出来。

上文已经提到，在日本是否能够实践本书的内容，决定了公司间的差距。各位读者也请学习、实践本书的内容，以使自己所在的公司能和竞争对手拉开差距。

如果您的实践一切顺利的话，请告知我一下，我们可以一起去喝一杯。只要有您的邀请，我随时都可以去中国！我非常喜欢绍兴酒，白酒也想尝试一下 :-)

作者代表　池田尚史
2014 年 11 月 15 日　于千叶县自家

译者序

《高效团队开发：工具与方法》并不是以实际的项目带你体验多人开发项目的整体流程，而是告诉你使用哪些工具和方法能够实现高效的团队开发。从版本管理系统、缺陷管理系统到 CI 工具、虚拟化、自动化测试等，无论你使用哪种语言、框架、软件开发模式，无论你是负责开发、测试，还是负责运维、项目管理，都会涉及这些工具。这些工具也直接影响着开发和运维的效率、项目成本以及公司的日常开销。

随着 SaaS（软件即服务）的普及，越来越多的项目已经不是经过一段时间的密集开发就结束的了。后期的开发，包括集成、测试、运维（部署、发布等），从重要性以及成本的角度来看都已经成为项目中的重要部分。本书后半部分介绍的持续集成、自动部署（持续交付）以及回归测试，都能有效地帮助这样的项目提高质量、加快开发速度、降低运维成本。

本书让我印象较深的一点是贯穿全书的自动化意识，包括自动化环境构建、持续集成、自动化测试、自动部署和发布。点击鼠标提交代码和测试用例，借助 CI 和各类自动化工具，自动触发编译、集成、测试、部署，还会自动将版本管理系统中提交的信息关联到缺陷管理和 CI 系统中，几分钟后打开浏览器就能够"享受"自己的劳动成果了。这样的场景实在太美了。想来是因为日本长期的劳动力不足以及高昂的劳动力成本才让作者对于自动化如此执着。对于还能够享受人口红利的中国软件行业来说，自动化也是非常必要的。除了能够在开发、测试、运维等多方面降低成本之外，自动化环境构建、自动化测试这样的机制能够降低项目对于成员个体的依赖，在大规模的团队开发中以及在灵活调整团队规模方面都是必不可少的。

本书所介绍的内容对于公司来说不仅可以提高效率，降低成本，还可以成为公司的一张名片。持续集成、自动化测试、持续交付，加上 Github、Jenkins、Vagrant、Chef、serverspec、Selenium 这些工具，由此构筑起的技术堆栈，无论是对于开发、测试人员还是运维人员来说都是

非常具有吸引力的。对于个人来说，除了扩展自己的知识之外，还能作为你选择公司的重要参考依据，判断公司是否对技术敏感，推测项目大致的工作流程以及是否可能成为 Death march。更重要的是

员工："老板，我要加工资！"

老板："为什么？"

员工："因为我长得帅！"

老板："……"

员工："因为我跟你 10 年了，没有功劳也有苦劳吧！"

老板："好吧，加 5% 差不多了。"

员工："这个项目交给我，我有办法只需要一半的人手就能完成！"

老板："真的？好！工资翻倍！"

最后感谢在翻译过程中给予我支持及鼓励的各位。特别是我的妻子，翻译这段时间恰好是她怀孕和生产的时候。我们平安地迎来了家里的新成员滚滚，借此祝愿他健康成长。

严圣逸

2015 年 3 月于上海

序言

本书名为《高效团队开发：工具与方法》。

读者朋友们大多都知道，团队开发是一件复杂、困难的事情。

关于团队开发，现在已经有了各式各样的方法论和工具。方法论方面，除了 Scrum、XP 等敏捷开发以外，还有些更具体的设计开发方法，如 TDD（Test Driven Development，测试驱动开发）、BDD（Behavior Driven Development，行为驱动开发）、TiDD（Ticket Driven Development，缺陷驱动开发）等，以及具体实践，如 CI（Continuous Integration，持续集成）、CD（Continuous Delivery，持续交付）等。讲述这些方法论的书籍、杂志以及网站到处都是，甚至多得让人不知该从何处着手。

再看一下从技术上支持这些方法论的工具，缺陷管理系统、版本管理系统、自动化测试、静态分析工具、自动化部署工具等，仅是种类就有很多。即使是简单地列举每一类中具有代表性的工具，其数量就多得令人感到头晕。

并且和数年前相比，支持团队开发的工具已经变得非常易用，能方便地构建高效的开发流程。但由于信息量过多，并且很分散，所以想要系统性地学习或者对新人进行高效的培训都还是比较困难的。正是因为意识到了上述这些问题，笔者才有了写作本书的想法。

本书以团队开发中所必需的工具的导入方法和使用方法为重点，对团队开发的整体结构进行概括性的说明。并且对“为什么用那个工具”“为什么要这样使用”等开发现场常有的一些问题进行举例说明。

希望你能喜欢本书。

读者对象

本书适合的读者对象有：

- 初次接手开发团队的项目经理
- 计划开始新项目的项目经理、Scrum Master

- **现有项目中返工、延期问题频发，想了解能帮助自己解决这些问题的工具及其使用方法的人**
- **测试是测试部门的事，与己无关；部署、发布是运维部门的事，与己无关——有如此想法的人**
- **想了解最近能够在 Web 服务开发中发挥作用的工具的人**

致谢

在写作本书的过程中有幸得到了诸多人士的帮助。在此特别感谢堀让治先生、冈村纯一先生、平田耕造先生、奥村康树先生、吉羽龙太郎先生、藤原大先生、中村知成先生、梅泽雄一郎先生、伊藤望先生、鹿糠秀俊先生、角田良太先生，他们极有耐心地审阅了我们粗劣的原稿，并提出宝贵意见。感谢 Vincent Driessen 先生和 Pablo Santos 先生提供的 Git-flow 图和 SCM 历史简图以及翻译许可。

还要感谢堀让治先生等想能（SHANON）株式会社的各位，在本书写作期间给予的各类帮助。

还有技术评论社的春原先生，从最初会面到现在的两年时间里，让您费心费力了。您一直极有耐心地等待着我那迟迟没有进展的原稿，实在是非常感谢。

最后要感谢我的妻子和两个孩子，对于休息日一直埋头写作的我，他们没有丝毫埋怨，给予了我百分之百的支持。还有陪着孩子玩耍的岳父岳母，以及我自己的父亲、母亲，谢谢你们！

作者代表　池田尚史

2014 年 3 月

目录

第 1 章

什么是团队开发

1.1 一个人也能进行开发

想必大家都知道，一个人也可以进行软件开发。自由软件以及共享软件的开发人员大多都是个人开发者。这几年来，以个人形式开发 iPhone 应用或 Android 应用并一炮而红的开发者也不在少数。网上也有一些个人开发的作为 OSS（Open Source Software）的实用小工具。

笔者从自己的兴趣出发，偶尔也会写一些 Jenkins[①] 和 Sublime Text[②] 插件与大家共享。一个人进行软件开发的情况下，因为没有沟通成本，所以能够迅速地开发并发布。在软件规模[③]较小的前提下，全部流程由一个人把控还是可能的。

但是如果软件的规模超过一定程度，仅由一个人把控产品的所有内容就比较困难了。不难想象，这时候就需要由多人组成团队进行开发。

而且人们对于软件产品的要求越来越高，因此软件开发，特别是在企业内部的开发现场，由多人组成团队进行开发是非常普遍的。

本书将由多人开发程序的体制称为团队开发。

① http://jenkins-ci.org/

② http://www.sublimetext.com/

③ 衡量软件规模的指标有很多种，例如用户数、功能数、代码行数等。

1.2 团队开发面临的问题

随着开发的进行，团队会遇到各式各样的问题：团队内部对遇到的问题没有共享、项目进度无法掌控、多人编写同一个产品的代码造成开发内容冲突等。而且由于涉及多个人，所以代码质量的统一以及产品代码的整体把控都变得越发困难。

如今的软件开发已经不是一旦发布（release）就意味着开发结束这样的模式了。很多情况下都需要在运营的过程中不断地更新。因此，在这样一段较长的时间中，有必要将一个人难以完成的代码交由多个人并行修改，并且要在一定程度上保证代码品质，防止退化（degrade）①，同时还要不断地增加新的功能。

人是会犯错并且容易遗忘的生物。微小的失误就可能造成系统的退化，并且对复杂的软件进行全方位、毫无遗漏的测试也是不可能的，而由多个人进行开发还有可能出现内容上相互矛盾的代码。即使是自己 1 个月前写的代码，也有可能完全忘记，甚至错认为是别人写的代码。

① 添加新功能或修正 bug 造成之前正常运行的其他功能无法运行或者运行速度变慢的现象。

1.3 如何解决这些问题

要解决团队开发所面临的问题，仅仅靠努力是行不通的。全世界的工程师们为了解决类似的问题而开发了各类工具和方法论，除了对其加以有效利用之外别无他法。

在多个人之间共享问题，将所发生的事情以简明易懂的方式公开，以及为了防止退化发生，执行自动化测试，这些都是非常重要的。除此以外，一旦发生错误，能够立即回滚到之前的状态也极其关键。另一方面，如果不能迅速地开发新功能并发布，就会在激烈的市场竞争中败下阵来，所以还必须并行地开发多个功能。当然这些都是以保证质量为前提的。

综上可见，团队开发绝非易事。为了团队开发能够顺利地推进，以下几点是必不可少的。

- “谁”“到何时为止”做了“什么事情”、“怎样”才算“完成”等，必须对这样的信息进行管理和共享
- 代码等各类工作成果，必须在团队内部共享
- 管理工作成果的变更，既要防止成果被破坏，又要保证各成员能够利用成果并行地作业
- 在团队中共享从项目中学到的知识
- 证明团队开发出的软件在任何时候都是可以正常运行的
- 构建任何人都可以正确开发、测试、发布的自动化工作流程

1.4 本书的构成

为了解决上述课题，本书将对各类工具及其使用方法进行讲解。

1.4.1 第 2 章：案例分析

在讲解的顺序上，首先为了了解诸事不顺、无法顺利进行的项目究竟是怎样的，我们将在第 2 章模拟在参与某个项目的两天时间内所发生的事情；然后介绍如何解决在这两天时间中出现的问题，并从之后的第 3 章开始对相应的工具进行讲解。

1.4.2 第 3～5 章：基础实践

从第 3 章开始，将依次介绍改善团队开发中需要进行的工作。各位读者可以按顺序阅读，并逐个实行。

第 3 章是关于版本管理系统（Version Control System，VCS）的内容。想必大多数人已经在使用版本管理系统了。本章将在回顾版本管理系统飞速发展的历史的同时，以 git-flow 和 GitHub-flow 这两个具有代表性的工作流程为例，对当下最流行的分布式版本管理系统的使用方法进行讲解。后面在内容上还更进一步，讲解数据库模式变更管理的重要性。

第 4 章是关于缺陷管理系统（Issue Tracking System/Bug Tracking System，ITS/BTS）的内容。这里的缺陷管理是指将完成项目所必需的任务（task）以 bug 票的形式进行管理。包括商用软件在内，缺陷管理工具的数量很多，让人不知该如何挑选，因此这里将介绍一些挑选时的要点。这一章还探讨了结合使用缺陷管理和版本管理系统所带来的优势，以及使用缺陷管理系统时课题的颗粒度问题。

第 5 章是关于持续集成（Continuous Integration，CI）的内容。估计很多开发现场都已经使用了版本管理系统和缺陷管理系统，但还没有使用 CI。进入测试阶段后，首次下载所有修改的代码并试着编译时，发现编译错误，或者程序无法启动，有如此经历的读者想必不在少数吧。

为了避免上述问题，就需要导入 CI。CI 会持续地下载代码的变更或者依赖关系的变更等所有变更内容并进行编译。导入 CI 可以同时提高开发速度及质量。这章将讲解实施 CI 所必要的编译工具、测试框架以及 CI 工具。

到第 5 章为止，都是在讲述为了顺利推进团队开发所需的基础实践。请认真阅读前 5 章，并试着实际操作一下。在前 5 章的内容已经全部实现的开发现场，如果要进一步推进团队开发，就需要用到从第 6 章开始介绍的内容。

1.4.3 第 6 ~ 7 章：持续交付和回归测试

第 6 章是关于持续交付（Continuous Delivery，CD）的内容。这里将对能够进行自动化环境构建，并结合 CI 实践在任意时间、重复地将运行的程序发布到正式环境中的机制进行讲解。

第 7 章是关于回归测试的内容。反复从用户角度运行用户验收测试，以此来确保没有退化发生，这是实现快速产品发布中必不可少的环节。本章将讲解使用 Selenium 和 Jenkins 来实现回归测试的自动化，还将进一步介绍为了加快测试速度所需要的分布式测试环境的搭建。

1.5 阅读本书前的注意事项

选择团队开发中所使用的工具和方法论，并没有唯一且绝对正确的答案。根据企业规模、产品规模、团队规模的不同，最合适的工具和方法论也不尽相同。而且，团队开发的工具、方法论的选择范围也非常大，但这并不意味着全都要使用。

1.5.1 最好的方法是具体问题具体分析

在团队开发的世界中，没有到哪都适用的万能的最佳实践，只有根据实际情况制定的方案才能行得通，这样的思考方法才是比较正确的。是否能找到最适合自己所从事项目的解决方案，可以说是能否将项目导向成功的关键所在。

例如，在实际的项目中，有需要将缺陷管理系统、版本管理系统、CI、CD 这些方法全部使用的，也有只需要版本管理和 CI 就足够的。

1.5.2 没有最好的工具

在工具的选择方面也没有正确的答案。仅以缺陷管理系统来说，是使用 Redmine[①] 好，还是使用 JIRA[②] 好呢？或者是只使用 GitHub[③] 就足够了呢？答案会因具体的情况不同而发生变化。

CI 的情况也一样，是使用 Jenkins 好还是使用 Travis CI[④] 好，或者是使用其他的工具比较好，不能一概而论。至于 CD，如今也有各类工具相继推出，哪个是最好的还很难说。

① http://www.redmine.org/
② https://www.atlassian.com/ja/software/jira
③ https://github.com/
④ https://travis-ci.org/

想要在本书中一一介绍各类方法和工具的所有组合模式是不可能的，所以本书将主要讲解本书写作时（2014 年 3 月）的一些优秀的模式。如果有多个选项的话，也将尽量向读者们展示。

在接下来的第 2 章中，我们将围绕诸事不顺、无法顺利推进的项目案例展开，一起思考该如何改善这种情况。

第2章

团队开发中发生的问题

2.1 案例分析的前提

用于高效推进团队开发的各类工具和方法数量众多。工具的话有版本管理系统（Version Control System，VCS）和缺陷管理系统（Issue Tracking System、Bug Tracking System，ITS/BTS）。方法的话有持续集成（Continuous Integration，CI）以及最近比较热门的持续交付（Continuous Delivery，CD）。

为什么需要这些工具和方法？为了回答这个问题，我们先要知道如果不使用这些工具和方法论，项目会怎么样。

本章将对陷入死亡行军[①]状态项目的两天时间进行案例分析，看一下类似这样的项目中容易出现的现象，随后反思其中存在的问题。并以此为基础，从第 3 章开始讲解具体的应对方法。

2.1.1 项目的前提条件

本章所涉及的项目的前提条件如下。

- **系统由网站 + 数据库构成**
- **不仅仅要进行开发，发布后还需要持续地进行版本更新和运维等**

各位可以想象一下自己身处自行开发并负责运维互联网服务的企业，或者负责开发企业内部业务系统的软件部门时的情形。

① 项目岌岌可危，开发人员身心俱疲的状态。

2.2 案例分析（第 1 天）

某天早晨你去公司上班，觉得这是一个郁闷的早晨，总想着“好讨厌啊”。为什么会觉得厌烦？因为你所参加的项目陷入了岌岌可危的状况，俗称“死亡行军”。

“但仔细想来，不是死亡行军的项目至今为止还没有遇到过”，你如此思考着，“可哪里出问题了呢？”你边想边去上班。

2.2.1 问题 1：重要的邮件太多，无法确定处理的优先顺序

你想着想着就到了公司，打开邮件客户端，便看到下面这样小题大作的邮件主题。

- 【重要】系统报错导致工作无法进行，请立即着手处理！
- 【加急】正式环境发生重大故障！麻烦立即处理。
- 【需要处理】客户环境发生重大退化。请尽快处理。
- 【正式环境故障】【重要】【加急】【立即处理】收到用户投诉。请立即处理。

看到这些五花八门的邮件主题，虽然你已经感到烦躁无比，但还是要着手确认邮件的内容。但是所有的邮件主题都标有【重要】或【加急】等字样，完全不知道应该从哪里着手，所以只能依次阅读收到的邮件并着手处理。

2.2.2 问题 2：没有能用于验证的环境

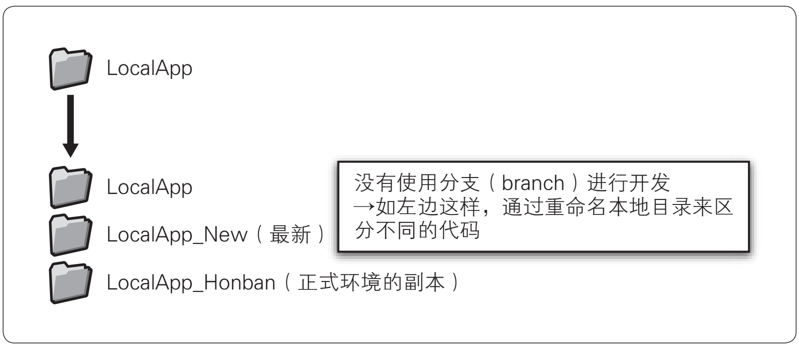
为了确认邮件中反映的问题，首先不得不构建和发生故障的正式环境相同的环境。因为这个项目一开始就没有准备用于验证的环境。

再现正式环境中发生的问题，必须首先再现与正式环境相同的环境。“要是有一直和正式环境保持同步状态[①]的验证用的环境，故障的再现就能简单多了”你一边想着，一边无可奈何地开始在本地机器上搭建和正式环境一样的环境。

2.2.3　问题 3：用别名目录管理分支

由于到昨天为止一直在开发新功能，自己本地机器上的代码和正式环境的代码已经相差甚远，所以只好将本地机器上的正在开发的代码所在的目录重命名为“XXX_new”，再从版本管理系统下载最新的代码。不这样的话，版本管理系统上的最新代码就会和本地的代码混淆（图 2.1）。

图 2.1　将目录重命名



其实公司从数年前就开始使用版本管理系统了，但由于开发人员中没有人精通版本管理系统的使用方法，所以并没有对其加以有效利用。

版本管理系统上经常只有最新的代码。那是因为没有有效地利用分支功能，为了区分本地机器上的代码和版本管理系统上的最新代码，只能用给目录重命名这种方式。合理地利用版本管理系统就应该能解决这些问题，但又不知道该如何操作。加上因为不知道标签（tag）的使用方法，发布时的系统快照也没能保存下来。而且版本管理系统上的最新代

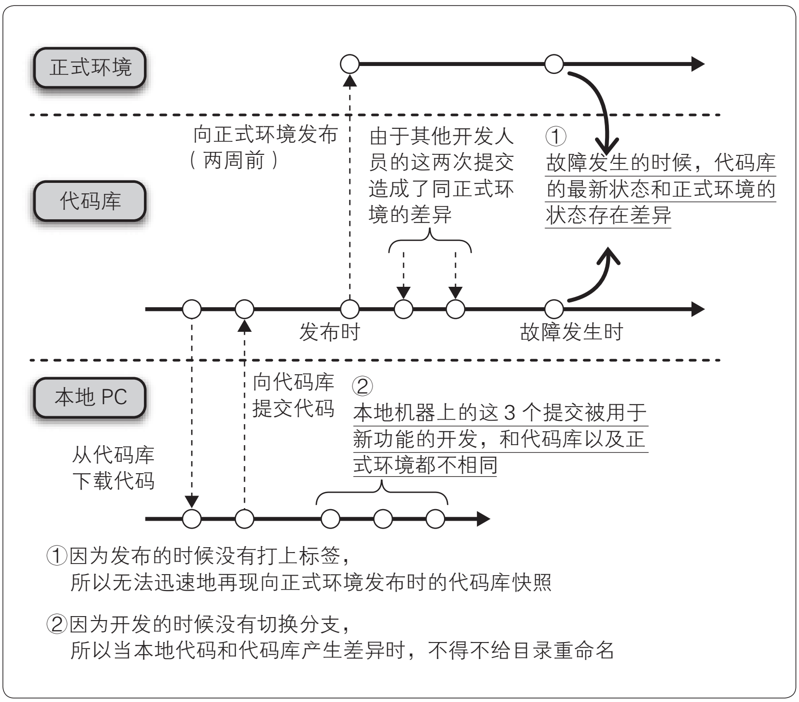
① 实际上要再现的环境包括服务器台数、网络构成以及数据库事务数据，但要将全部内容都做得和正式环境完全一致是非常困难的。本书中指的是应用程序层面上的再现，也就是指程序的代码、数据库模式（schema）、中间件的配置等。

码还有可能已经和正式环境的代码有了差异，毕竟向正式环境进行发布已经是两周前的事情了。

因为没有使用标签，所以正式环境发布的版本对应版本管理系统中哪个版本[①]已经无从知晓。没有办法，只能麻烦负责运维的人将正式环境的代码取了下来。

自己试着比较了最新的代码和正式环境中的代码，的确有所差异[②]。在最新代码的环境中，故障似乎已经无法再现（图 2.2）。

图 2.2　正式环境和开发环境产生差异



① 这里使用了“版本”这个词，在 Subversion 中指的是“版本号”（revision），在 Git 中指的是“提交哈希”（commit hash）。各版本管理系统的叫法不相同，所以这里统称为“版本”。

② 这个例子假定的是 PHP 等脚本语言。Java 或 C# 的情况下，除了 Play Framework 等一部分情况例外，通常部署的都是编译后的二进制文件，无法简单地进行比较。

无法再现故障的代码就没有意义，所以你决定继续使用从正式环境获取的代码。用这份代码在本地机器上构建环境可不是一件容易的事情。还必须保证数据库模式和正式环境完全相同，而重新制作数据库也是一件非常麻烦的工作。

2.2.4 问题 4：重新制作数据库比较困难

为了使数据库和正式环境完全相同，需要下载版本管理系统的代码库中的 SQL，并在自己的开发环境上构建数据库。刚才还在使用的用于新功能开发的数据库的模式已经和正式环境的数据库模式不一样了。仔细看了下下载下来的 SQL，总觉得有一些 SQL 并不适用于开发环境。

不知道该执行哪些 SQL 才能构建和正式环境完全相同的数据库。是否有必要记录下执行了哪些 SQL 呢？没办法，姑且只能根据推测执行 SQL，想办法试着构建环境了。也不知道是否能正确地再现正式环境，不过也只能这样了。

数据库构建完后，为了再现和正式环境相同的状态，还需要导入数据。可是在将正式环境的数据[①]导入刚才构建的数据库时，不知为何部分数据的导入失败了。应该是数据库模式不同的原因吧。总觉得正式环境的模式和刚才在本地构建的模式有所差异。虽然不清楚是为什么，但鉴于时间也所剩无几了，只能自己用眼睛来寻找差异，并修改 SQL，然后重新构建数据库了。重新导入数据，这次终于成功了。

代码和数据库都准备好后就可以试着运行程序了。赶紧依照报告中的描述操作一遍，却发现程序无法正常运行，并且和报告中描述的现象也不一样。

为了再现故障而试着搭建环境，但因为出现了其他问题，结果以失败告终。恐怕还是因为数据库模式和正式环境不一样吧。以为仔细点就能避免上述问题的，结果还是不行。没办法，只能将正式环境的数据库

① 这里的数据是指正式环境的主数据（master data）。关联数据（transaction data）因为数据量过大，应避免直接导入本地环境。

原封不动地复制到开发环境了[①]。

终于搭建好了和正式环境完全相同的环境，突然发现已经是中午了。已经过去了两个多小时，可是连邮件中反映的问题是否能再现都还没确认。

当天下午和同事们分头把早晨 4 封邮件中的问题全部再现了，用了 1 天的时间修正这 4 件 bug，并将其提交到版本管理系统。做完这些已经快要到末班车的时间了。今天一整天都在忙着处理这些问题，本来应该进行的新功能开发也完全没有着手。算了，明天再干吧，回家了。

① 这个例子中，正式环境的数据量并不是太大，所以可以复制。如果是大规模服务的话，复制是不现实的。为了避免上述问题，需要考虑数据库的管理。

2.3 案例分析（第 1 天）中的问题点

看了死亡行军项目中的 1 天，感觉怎么样？“项目一直都是这样的，已经习惯了”是不是也有人这么想呢？

反思第 1 天中发生的事情，让我们一起来确认下到底哪里有问题，哪里做得不够好。

2.3.1 问题 1：重要的邮件太多，无法确定处理的优先顺序

这个项目中没有对课题进行管理的机制（缺陷管理系统），所以只能通过邮件来发送有关 bug 或故障的报告。加上邮件的主题中写有【重要】【加急】【需要处理】这样的标题头，所有的邮件都变得很重要，不知道应该从何处开始着手处理。这样的邮件满天飞的开发现场你有没有待过呢？“遇到过遇到过”似乎还能听到这样的回答。

不使用缺陷管理系统，而通过邮件进行交流会有哪些问题？下面就来列举一些。

●……邮件的数量太多，导致重要的邮件被埋没

大家在开发现场 1 天之中所收发的邮件数量大概有多少呢？当然数量会根据公司的规模和职位有所不同。以笔者为例，笔者之前所在的公司中，与自己所参与的项目相关的邮件以及其他邮件加起来，1 天之中大约要收到 300～400 封邮件。该公司的规模未满 100 人，参与项目开发的人员在 10 人左右。笔者当时还兼任部门主管，所以会有人事相关的邮件以及其他的一些被抄送的邮件，这也是邮件比较多的原因之一。

也就是说，参加项目的人员不可能都专门进行这一个项目，其中很多都还有其他的日常业务，或者兼任着其他项目的工作。因此必然会产生大量的邮件，造成重要的邮件被埋没，容易被漏看。

为了让自己的邮件不被漏掉，很多人都会在邮件的主题前加上之前列举的那些标题头。但是因为没有统一的规则，究竟哪些是重要的无从知晓。结果还是被埋没在了成堆的邮件中，添加的标题头也完全失去了意义。

●········ 无法进行状态管理

因为邮件并非是用来管理课题或任务的软件，所以不能对其状态进行管理。哪个课题已经结束了，哪个课题还在进行中，或者是验证不通过被退回来了，这些都无法简单地获知。如果不耐心地一封一封地阅读同一主题的往来邮件，就不知道究竟是怎样的问题。有时就算读到了最后，因为来来回回的交互过于错综复杂，究竟结局怎样也还是不清楚。这也是常有的事。

●········ 直观性、检索性较弱

这一问题同上一个问题是相关的，那就是用邮件进行管理在直观性和检索性方面也比较弱势。邮件中包含了很多和项目没有直接关系的内容，想看只包括课题的列表，这样的要求估计无法实现。检索也是这样。也可以使用像 Gmail 这样的邮件应用程序，但像这样用全文检索来查找课题可以说实在是效率太低了。

●········ 用邮件来管理项目的课题

总结一下上述内容就是：邮件的可视性不佳，难以进行优先度或重要度的判断，并且无法进行状态管理，不支持显示课题列表、检索等，所以作为管理项目状况的工具来说，功能上是不足的。

项目混乱是由于某些问题造成的，其中一个比较严重的问题就是这里列举的没有共享信息及有效管理课题的机制。

上述问题可以考虑通过导入缺陷管理系统来解决。本书的第 4 章将讲解在使用了缺陷管理系统的项目中任务管理和 bug 管理的方法。

2.3.2　问题 2：没有能用于验证的环境

没有能用于验证的环境可以说是一个大问题。在收到项目正式环境中发生的故障报告后，下载代码并搭建开发环境的做法每次都要花费较长时间。

只要不影响新功能开发的任务不就没事了？但现实往往没这么简单。除了发布后的正式环境之外，一般还有为下一次发布做准备的，还在验证中的 staging 环境[①]，甚至还有为下下次发布准备的环境。

新功能开发过程中有时不得不调查正式环境中发生的一些故障。如果有一直和正式环境保持一致的 staging 环境那就最好了，但因为环境的搭建实在是件非常麻烦的事情。要怎样做才能提高效率呢？

有方法能够搭建持续、自动地向验证环境、staging 环境以及正式环境进行发布，并一直保持正常运行的环境，本书的第 6 章中将进行这方面的讲解。

2.3.3　问题 3：用别名目录管理分支

这个项目虽然使用了版本管理系统，但不能说对其进行了有效的利用。以前连版本管理系统都不使用的开发现场随处可见，现在使用版本管理系统的开发现场比较多了。但是如同这个项目一样，版本管理系统没有得到充分利用的情况却意外地多。

使用版本管理系统是为了管理什么时候、谁、做了怎样的修改（提交记录的管理），以及能恢复到过去某个时间点的状态（分支、标签的管理）。这个项目中没有很好地使用版本管理系统的重要功能——分支和标签，所以无法从新功能开发的版本顺利地切换到有故障报告的已经

① 这里的 staging 环境是指介于开发环境和正式环境之间的测试环境，即正式环境发布前用于验证的环境。本书中将 staging 环境用于再现和验证在正式环境中发生的故障。根据环境的体系不同，也有和正式环境保持完全一致，仅在发布前用于动作确认的 staging 环境。在这样的情况下，除了 staging 环境之外，还会另行准备测试环境。环境搭建包括运维在内是一笔不小的开销，所以在体系以及项目预算允许的范围内，最合理地构建环境是很重要的。

发布的版本。只能临时将本地机器上的目录重命名。也就是说，本来应该交由版本管理系统负责的分支管理，现在由人在本地机器上手动地进行了。

读过前文的读者想必都知道了，用给目录重命名的方法修改完 bug 后回到新功能的开发，之后再一次回到 bug 修改，在这样反复地进行重命名的过程中，很容易出现问题。

那么到底该如何使用版本管理系统呢？版本管理系统的分支、标签这样的功能如何使用才是有效果的呢？

这个问题还关系到今后程序以怎样的周期、怎样的方式进行发布，本书将在第 3 章进行这方面的讲解。

2.3.4 问题 4：重新制作数据库比较困难

在构建 staging 环境或正式环境时比较容易出问题的是数据库的处理。虽然没有对数据库的修改进行管理的项目比较常见，但这个项目还是使用版本管理系统对 SQL 文件进行了管理的。

这样就会经常出问题。比如不知道在什么环境中该使用哪些 SQL，从而导致没有察觉到遗漏了某些 SQL。还有在多个人修改数据库的情况下，不知道该如何管理才能避免修改冲突。

那么这个项目到底会怎么样呢？

首先，版本管理系统上，有如下的 SQL 被提交。

```
版本管理系统上：
Rev1:
提交者： ikeike443
create_user.sql
  CREATE TABLE User(
    id SERIAL NOT NULL,
    name VARCHAR NOT NULL,
    email VARCHAR NOT NULL
  );

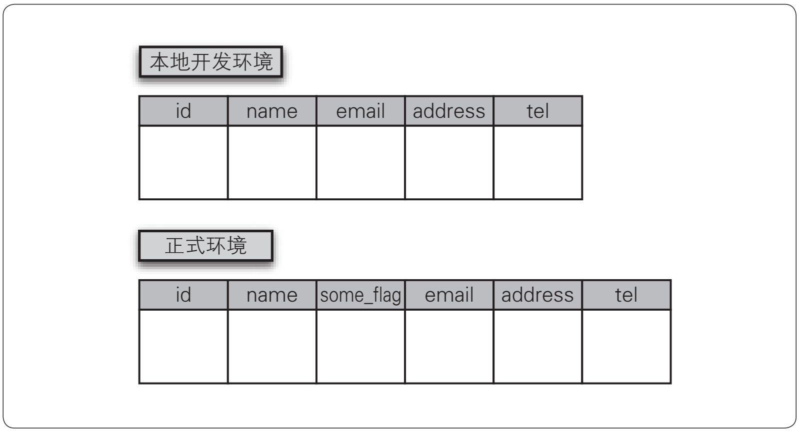
Rev2:  本地环境中遗漏
提交者： hori
add_column.sql
  ALTER TABLE User ADD COLUMN some_flag INTEGER;
```

```
Rev3:
提交者: ikeike443
user_add_address.sql
  ALTER TABLE User ADD COLUMN address VARCHAR;

Rev4:
提交者: okamura
tel_add.sql
  ALTER TABLE User ADD COLUMN tel VARCHAR;
```

对应上述 SQL，本地开发环境和正式环境的模式如图 2.3 所示。

图 2.3　本地开发环境和正式环境的模式



ikeike443 的本地环境中遗漏了 Rev（Revision）2 的 add_column.sql，只执行了 create_user.sql 和 user_add_address.sql。但因为没有注意到这点并继续执行了之后的 tel_add.sql，所以就和正式环境产生了差异。

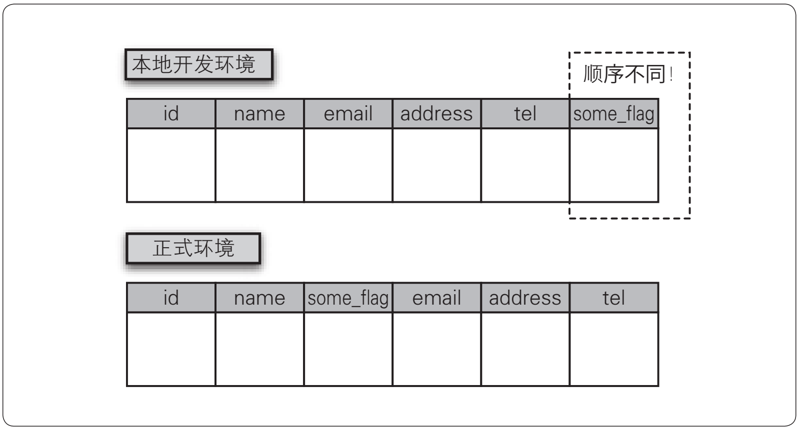
上述例子中是用眼睛看出数据库模式上的差异后，手工制作了相当于 add_column.sql 的文件并执行的。但结果还是出现了原因不明的运行时错误。于是只好放弃，转而从正式环境复制所有的数据重新构建环境。

这个谜一般的运行时错误的原因是什么呢？

这是因为本地开发环境和正式环境中 ALTER 语句的执行顺序不同，造成了列的定义顺序和最新代码不匹配（图 2.4）。根据代码的写法以及对象关系映射（O/R mapping）的实现的不同，有时会出问题，有时也可能没有问题。但关键问题在于因为没有制定构建数据库的方法，所以

无法构建出能够保证正常运行的环境。

图 2.4　执行的顺序不同



要怎么做才能解决这样的问题呢？

第 3 章中将讲解用版本管理系统管理 SQL 的方法以及处理数据库变更管理的工具。

2.4 案例分析（第 2 天）

接着来看一下项目第 2 天中发生的事情。项目会怎么样呢?

2.4.1　问题 5：不运行系统就无法察觉问题

第二天早上，刚坐到椅子上准备继续开发新功能时，测试人员来到了你的座位前。说是昨天提交的版本有问题。在提交的 4 个 bug 中，能够确认其中 3 个已经得到了修正，但另外 1 个还是有问题。

并且还有报告说发生了退化，以前修正好的 bug 因为这次的修改又再次出现了。可昨晚明明自己和组员一起努力把邮件中提到的 4 个 bug 都修正并提交了啊，而且还在各自的环境中对修正进行了确认呢。

不过重新回想一下的话，确实没有将全员的代码集中到一起运行过。因为在测试人员的环境中是第一次将所有的修改合并到一起运行的，所以就出现了上述状况。

2.4.2　问题 6：覆盖了其他组员修正的代码

下载最新的代码并运行，发现确实如测试人员所说：3 个修正了，1 个没有修正，并且过去修正了的 bug 又复活了。你觉得实在是很奇怪，就看了下代码库的提交记录[①]。

```
rev: 245
Author: ikeike443 <ikeike443@gmail.com>
Date:   Mon Dec 24 23:59:59 2012 +0900

    修正了发送邮件的逻辑

rev: 244
Author: okamura <hogehoge@gmail.com>
```

① 这里以 Subversion 为例。

```
Date:   Mon Dec 24 21:04:57 2012 +0900

    修正了申请时扣款处理失败的问题

rev: 243
Author: ikeike443 <ikeike443@gmail.com>
Date:   Mon Dec 24 19:55:55 2012 +0900

    修正了某些时间点无法进行申请处理的bug
```

Rev244 和 243 都修改了申请处理相关的部分，觉得这里有些奇怪，就查看了下 Diff[①]，才发现你在 Rev243 中提交的修改被 Rev244 覆盖掉了。

你所提交的修改如下所示[②]。

Rev243和Rev244的Diff（在Rev243中增加的修改）

```
- if(user.status == 3) {
-   application.submit()
- } else {
+ //考虑到用户状态是Null的情况
+ if(user.status != null && user.status == 3 ) {
+   application.submit()
+ } else {
```

这个修正被之后的提交覆盖，如下所示。

Rev243和Rev244的Diff（在Rev244中增加的修改）

```
-  //考虑到用户状态是Null的情况
- if(user.status != null && user.status == 3) {
-   application.submit()
- } else {
+ //使用信用卡的用户的情况下，在申请的同时进行扣款处理
+ if(user.status == 3 && user.useCredit == true) {
+   application.submit();
+
+   billing.creditStatus = 1;
+   billing.execute();
+
+ } else {
```

① 原指比较文件并输出文件之间的差异的程序。这里指差异本身。

② 这段代码自身原本就有如下列举的这些问题。有必要从根本上提高代码的质量。

- 使用了魔数（magic number）
- 没有确认 application.submit 的返回结果
- 没有处理异常
- 代码的实现有副作用（side effect）

在 Rev243 中特地加上的 user.status 的 Null 检查[①],被 Rev244 覆盖后 Null 检查就没有了。为什么会发生这样的事情呢?

结果，原以为已经修正的 bug 还是有问题，原因是代码被覆盖而造成了退化[②]。向覆盖代码的开发者询问事情的缘由，对方却只是回答道："向代码库提交在自己的本地环境中修正的代码时发生了冲突（conflict）[③]，我只是把冲突改掉了。"

这时你很想斥责对方："那是你修正冲突的方法有问题!"但还是控制住了自己的脾气，包括那个开发者的修正在内，你对代码做了如下修正。

Rev245和最新的Rev246的Diff（这次的修正）

```
- //使用信用卡的用户的情况下，在申请的同时进行扣款处理
- if(user.status == 3 && user.useCredit == true) {
-   application.submit();
-
-   billing.creditStatus = 1;
-   billing.execute();
-
- } else {
+ //使用信用卡的用户的情况下，在申请的同时进行扣款处理
+ //考虑到用户状态是Null的情况
+ if(user.status != null &&
+    user.status == 3 && user.useCredit == true) {
+   application.submit();
+
+   billing.creditStatus = 1;
+   billing.execute();
+
+ } else {
```

2.4.3　问题 7：无法自信地进行代码重构

这么修改姑且合并（Merge）[④]成功了，但因为 if 语句发生了变化，所以程序的动作也会发生变化。这时你觉得进行一下代码重构比较好，但是却没有信心能够在确保不发生退化的情况下进行代码重构。没有办

① "user.status != null"这一部分。可见在 Rev244 中被删除了。

② 由于添加功能或者修改 bug 而造成其他已经实现了的功能无法运行或速度变慢的现象。这次出现的现象是，因为 Rev244 的修正使得 Rev243 中修正的代码被复原了。

③ 对同一处代码进行了不同的修改，造成了代码修改冲突。

④ 将多件物品整合到一起。

法，只能增加 if 条件，用条件分支来处理。

Rev245和最新的Rev246的Diff（重写了这次的修正）

```
- //使用信用卡的用户的情况下，在申请的同时进行扣款处理
- if(user.status == 3 && user.useCredit == true) {
-   application.submit();
-
-   billing.creditStatus = 1;
-   billing.execute();
-
- } else {
+ //使用信用卡的用户的情况下，在申请的同时进行扣款处理
+ //考虑到用户状态是Null的情况
+ if(user.status != null &&
+    user.status == 3 && user.useCredit == true) {
+   application.submit();
+
+   billing.creditStatus = 1;
+   billing.execute();
+
+ //不使用信用卡的用户的情况下
+ } else if(user.status != null && user.status == 3 ) {
+   application.submit()
+ } else {
```

至此昨天发生的 4 个 bug 应该都修正好了。为避免测试人员再像今天早上那样不给你好脸色看，所以重新对所有的用例进行测试。没有准备自动测试环境，只能手动进行测试。测试结果没有问题，程序也能正常运行，所以就把代码提交了。

2.4.4 问题 8：不知道 bug 的修正日期，也不能追踪退化

还有一件事情：过去的 bug 又出现了。这究竟是怎么回事呢？向测试人员问了下这个 bug 原本是什么时候发生的，具体是怎样的 bug，回答说是大约半年前直接收到客户的邮件后，让其他开发人员修正了的 bug。直到现在才第一次听说这件事情。过去修正了的 bug 再度发生，可以说是发生了退化，当然不可能就这样进行发布。

没办法，只能让测试人员把过去的邮件翻出来，找出和用户交流的内容。根据好不容易才找到的邮件日期，在版本管理系统的代码库中查找对应的提交版本，再和当时负责修改的开发人员一起来确认，终于找

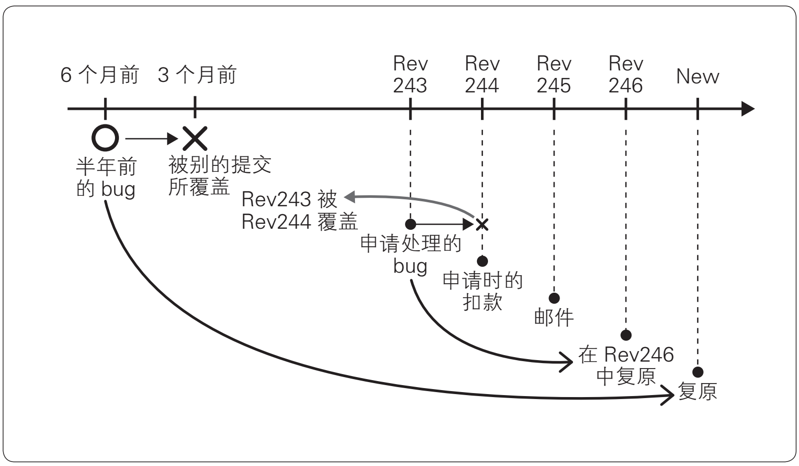
到了当时的提交版本。

但似乎这个提交在大约 3 个月之前就被其他的修正发布覆盖掉了。正式环境发生退化长达 3 个月之久，期间谁都没有察觉出来，这实在是非常糟糕的情况。这次能够在客户提出之前发现，实在是不幸中的万幸。

重新修正并提交。想着这下应该没问题了，但还是觉得不安心，于是再次手动进行了测试，确认退化已经被修正了。

啊！差点忘了，昨天发生的 4 个 bug 的修正没关系吧。修正了过去的退化，却导致了别的退化发生，这可不是闹着玩的。想到这里，再次对 4 个 bug 进行了测试。这次确实没问题了，于是提交了代码（图 2.5）。

图 2.5　至今为止的提交状况



2.4.5　问题 9：没有灵活使用分支和标签

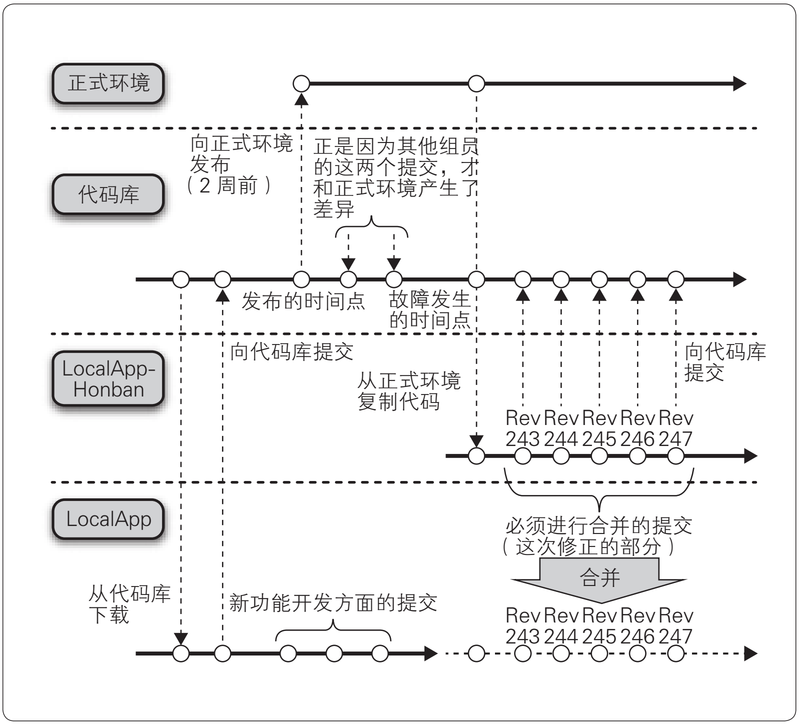
在这样那样的“斗争”中，不知不觉已经是傍晚了。本来应该做的新功能开发还没有开始动手。总觉得每天都在干同样的事。

不管怎么说总算告一段落了，该回到新功能的开发上了。用昨天重命名的目录，重新开始开发。

稍等一下，从昨天开始斗争了两天的 bug 修正会怎么样？如果就这样在昨天的目录的基础上进行新功能开发，并提交到版本管理系统的代

码库中的话，感觉还是会发生大范围的退化[1]。真是太险了。你注意到了这点，开始将昨天的修正合并到手头的新功能开发版本中（图 2.6）。幸运的是合并操作可以使用工具机械地进行，但编译却出了问题[2]。

图 2.6 没有忘记对修改进行合并，但 ......



一边发愁一边继续修改，终于编译是能通过了，但确认昨天之前的程序动作是否正常时却遇到了困难，因为昨天的事情几乎已经不记得了。总算编译也通过了，程序也能运行了，觉得应该没有问题，于是就着手继续新功能的开发了。

虽然也觉得正是因为反复出现这样的事情才导致了现在的状态，但

① “如果合理使用 Subversion 或 Git 等版本管理系统的话就不会发生这样的事情”，你的这一想法是正确的。

② 聪明的读者可能已经注意到了，因为忘记了合并图 2.6 的“代码库”部分中的“别的组员的两个提交”，所以发生了编译错误。没有合理地对分支进行管理，就会发生这样的事情。

却不知道应该怎么做。就这样怀揣着说不清的不安，继续回到了工作中。

2.4.6 问题 10：在测试环境、正式环境上无法运行

夜幕降临，新功能的编码工作终于步入了正轨，这时测试人员发来消息，说刚才提交的版本在 staging 环境的特定条件下无法正常运行。没办法，只好和测试人员一起对 staging 环境进行了确认。

的确，程序自身启动起来了，过去 bug 的退化也确认修正了，但其他的功能却出现了“Internal Server Error”[①]。刚才明明在本地确认过了，为什么还会出现错误呢？

于是决定在本地环境上确认同样的操作。但由于现在本地机器上的开发环境是新功能开发用的环境[②]，因此只能再一次重命名目录，回到刚才还在使用的修正 bug 用的目录中进行确认。

在本地机器上确认动作，没有发现问题，可以正常运行。应该是本地环境和 staging 环境有什么地方不一样吧。这时突然想了起来，昨天修正 bug 的时候添加了新的库，而 staging 环境上还没有安装过新添加的库。

的确，在本地确认完后，忘记让测试人员安装库了。用邮件将本地安装的库发送给测试人员进行安装后，staging 环境能正常运行了。总算松了口气，但觉得这么下去不是个办法，难道没有更有效的方法来管理库，并且能防止遗漏吗？

2.4.7 问题 11：发布太复杂，以至于需要发布手册

总算 staging 环境上的测试也 OK 了，准备向正式环境进行发布。由于发生了之前遗漏库的问题，负责发布的人员要求你提供发布手册。于是你就和测试人员一起制作了发布手册，并在制作过程中特别留意了以下几点。

---

① 网站服务器内部发生的错误。

② 为了保险起见进行确认时，发现该项目中没有用于验证的环境，所以只能在本地机器上通过目录重命名的方式来验证。

- **应该下载版本管理系统的代码库中的哪个版本**
- **应该如何更新 DDL、依赖的库以及配置文件**

每次都需要写这样的手册。有没有什么更高效的做法呢？

因为这次的发布中包括重大的 bug 修正，作为开发者的你被要求一同在场。每次发布都要花费一天的时间，彻夜进行。虽然你有些不情愿，但也没有办法。今晚看来是回不去了。

在彻夜的发布作业中问题频发。正式环境的数据库由于经历了至今为止的开发过程，已经和测试环境以及 staging 环境有了很大的差异，只能通过手动修改来处理。配置文件也是只有正式环境的写法不一样，不得不一边谨慎地用肉眼核对，一边进行合并。

所依赖的库的版本不一样的情况也有很多。为了让正式环境运行起来，需要很多额外的作业，并且这些作业的内容没有被记录在版本管理系统中，所以恐怕下一次发布的时候还是会发生同样的问题。

就这样，死亡行军仍在继续。

2.5 案例分析（第 2 天）中的问题点

让我们一起来反思一下第二天中发生的事情。

2.5.1　问题 5：不运行系统就无法察觉问题

这个例子中到底发生了些什么呢？

前一天几个人分头修正了 4 个 bug。但第二天早上全员的修正代码合并到一起测试时，4 个 bug 中只有 3 个被修正了，并且过去的 bug 还发生了退化。如果开发人员没有说谎的话，那么就有可能是修正的内容互相发生了干扰。

在这个例子中，版本管理系统使用方法不正确是造成他人的修改内容被覆盖并消失的主要原因。仔细地跟踪调查版本管理系统中的提交记录，就能够彻底地弄清原因。

但这里的问题是：到发现问题时，已经过去了一天一夜。等到发现过去修正的 bug 的退化，那经过的时间就更长了。

假如因为几个月前的修正而发生了退化，那么追查提交记录并查明原因大概需要多久的时间呢？请试着想象一下。这时你可能连想都不愿意想，更别提去做了。难道就没有办法早一点发现这个问题吗？

每次向版本管理系统提交更新时，都对程序整体是否能正常 build、已有的功能是否正常运行进行检查不就可以了吗？

这样的想法称为持续集成（CI）。这是将团队成员的修改等所有项目相关的资源集中到一起进行集成，并经常、持续地确认 build 及测试是否通过的一种实践。

CI 相关的内容将在第 5 章中进行讲解，用 CI 实现高效的回归测试相关的内容将在第 7 章中进行讲解。

2.5.2　问题 6：覆盖了其他组员修正的代码

多人开发程序时会发生因修改的代码重叠而产生冲突的现象。发生冲突的情况下，要在保证双方修改都能正常运行的前提下进行合并，但要正确地进行合并是非常困难的。并且无法保证每个团队成员都能进行合并，偷偷地直接将他人的修改覆盖而不进行合并也完全有可能不被发现。

Subversion 和 Git 等比较新的版本管理系统的设计思想是：原则上不对文件加锁，并对多人的修改进行合并处理，有冲突时会明示并让冲突发生。

另一方面，以前的 VSS（Visual Source Safe）[①]等版本管理系统的设计思想则是对文件加锁来避免冲突的发生。

如果团队中有工程师是在使用 VSS 等基于锁的版本管理系统的开发现场成长起来的话，可能会因为不习惯 Subversion 和 Git 这类基于合并的管理系统的思维方式，而对于发生冲突时一定要消除冲突无法理解。也可能是因为开发人员觉得无视冲突，强行覆盖原有代码不会有什么问题，结果造成了类似这种现象的发生。

关于使用版本管理系统来消除冲突的方法，以及基于合并的版本管理系统更为优秀的原因等，将在第 3 章中进行讲解。

在类似于这次例子的情况中，如果写好测试用例并用 CI 进行测试的话，应该就能及早发现问题。CI 相关的内容将在第 5 章进行讲解。

2.5.3　问题 7：无法自信地进行代码重构

几处修改 bug 的代码因被其他组员的提交所覆盖而消失了。通过追查提交记录终于把问题搞清楚了，但怎么修改才是最优雅的，成了件烦心的事情。

在多人进行的开发中，修改的地方发生冲突是因为在同一处地方基

① 微软在 2012 年之前提供支持的版本管理系统。现在提供的是其后续产品 Team Foundation Server。

于不同的目的添加了不同的代码。这意味着需要以某种形式来重新考虑代码的构造，也就是说需要进行重构。

根据维基百科，重构的定义是这样的。

> 重构（refactoring）是指计算机编程中，在不影响输出结果的前提下对代码内部的构造进行整理。

正如上面所描述的“不影响输出结果”，重构必须保证程序的对外输出保持不变。也就是说需要定义出该程序中什么是正确的。

方法之一就是准备好规格说明等资料。这个方法的确可以证明“正确性”，但每次都要手动确认重构后程序的动作是否符合规格要求，实在太耗费时间了。

一想到又费时又麻烦，心理上对于重构的抵触情绪就愈发高涨，不愿意动手去做。在一些开发现场，“不要动已经在运行的程序”像这样明令禁止重构的情况也是存在的。

为了消除这样的抵触情绪，能够自信地进行重构，测试代码的编写就显得尤为重要。如前所述，重构是“在不影响输出结果的前提下对代码内部的构造进行整理”，因此只要编写的测试代码可以保证输出结果不发生变化就可以了[①]。将测试代码做成只调用一个命令就能执行的形式，这样就可以简单地反复进行测试，从而在任何时间都可以迅速地对程序的正确性进行确认。只有有了这样的测试环境，才能重新着手重构工作。

能够为编写测试代码提供方便的测试框架有很多。测试框架以及测试代码的写法将在第 5 章进行讲解。

◆◆◆

成功编写测试代码后，心理上对于重构的抵触情绪就能大幅减少。在进行代码重构后并提交到版本管理系统的代码库之前，调用一条命令执行测试，这样就能对重构内容的正确性进行确认。所以即使重构中发

① 最好是对对象类中的 public 方法，即公开的 API 编写单元测试。但是由于类的分割不合理、数据库或用户界面和系统的耦合过于紧密等原因，无法编写单元测试的情况也时有发生。在这样的情况下，可以对系统最外侧的 API（一般情况下是用户界面）编写测试代码。相关内容将在第 7 章中进行讲解。

生了错误，也能在提交之前及时发现。

进一步导入 CI，让测试代码能够一直自动执行。对程序正确性进行测试的机会越多，越能够安心地进行重构。即使在本地环境中通过测试，和其他开发人员修改的代码合并后仍有可能测试失败。并且开发人员毕竟也是人，所以在提交之前忘记执行测试也是有可能的。导入 CI 的话，因为 CI 服务器会自动执行测试，所以就能够及时发现问题。CI 服务器可以每天定时执行测试，也可以每当向版本管理系统的代码库进行提交时执行测试，根据配置可以在各种时间点执行测试并发现问题。通过编写测试代码以及导入 CI，终于可以自信地进行代码重构了，产品的品质也自然而然地有了提高。

相反，既不写测试代码，也不进行 CI，并且也不进行重构，这样持续维护的代码就会成为巨大的负担，直至阻碍事业的发展。这样的代码在《修改代码的艺术》[①]中被定义为 Legacy code。

不编写测试代码导致产生大量的 legacy code，因此软件的品质完全无法提高。确认“正确性”的手段只有手动和用眼睛看，在这样的情况下，“正确性”的确认就会白白浪费大量时间，执行回归测试[②]就更不现实了。越没时间越不写测试代码，从而产生越来越多的 legacy code，这样便陷入了恶性循环之中。

这样的恶性循环持续几年后便会陷入绝境，不要说添加新功能了，连 bug 修正都忙不过来，最终只能被时代所淘汰。这样的软件产品并不在少数。

关于重构所必需的测试代码的写法，以及持续地自动进行测试的 CI 实践，这些将在第 5 章中进行讲解。

2.5.4　问题 8：不知道 bug 的修正日期，也不能追踪退化

在刚才的例子中，发生了以前修正的 bug 再度出现（发生了退化）的情况。不知道 bug 是什么时候发生的、是怎样的 bug、在哪次提交中

① 《修改代码的艺术》（美）Michael C. Feathers 著，侯伯薇译，机械工业出版社 2014 年出版。——译者注

② 该测试的目的是检查程序的修改所带来的影响。具体请参考第 7 章。

被修正，这样的事情在已经运营较长时间的系统中可能是比较常见的。如果仅用邮件或口头交流故障和 bug，没有在团队成员之间共享信息，就容易发生这样的事情。为此，首先使用缺陷管理系统将问题从发生到解决的所有过程记录下来是非常重要的。关于缺陷管理系统，将在第 4 章进行讲解。

然后，通过使版本管理系统和缺陷管理系统进行交互，就能关联代码的修改记录和问题票，并记录下来。这样就可以从问题票追踪到代码的修改记录，找出 bug 是何时修正的、谁修正的、如何修正的这些信息。反过来也可以从版本管理系统上的修改记录追踪到描述问题的 bug 票。如此一来，就既可以从问题票追踪代码，查看代码被做了怎样的修改，即过去的问题票的处理结果，又可以从代码的提交记录追溯问题票，查看问题的原因，使双向追踪成为了可能。

版本管理系统和缺陷管理系统高效进行交互的方法将在第 4 章中进行讲解。只需稍微花一些功夫，问题的可追踪性[①]就会有所提升，这是非常有效的实践。

并且，CI 和缺陷管理系统以及版本管理系统这三者之间的交互也是非常重要的。这样一来，某个问题是在什么时候被什么人怎样修改的，以及修改结果是否通过了测试、是否反映到了 staging 环境、是否发布到了正式环境等，整个过程就都可以进行追踪。CI 和缺陷管理系统以及版本管理系统的交互，可以毫不夸张地说是现代系统开发中的三种神器。特别是在实行敏捷开发的情况下，这些是最基础的实践项目。关于这部分将在第 5 章中进行讲解。

更进一步，如果和部署自动化工具相关联，那么到部署、发布为止就都可以进行统一的管理。近年来，一些最先进的开发现场已经在尝试自动化部署。包括自动化部署在内的管理相关的内容将在第 6 章进行讲解。

① 追踪的可能性的意思。

2.5.5 问题 9：没有灵活使用分支和标签

这里举的是修正结束后回到新功能开发时，差点忘记合并的例子。问题 3 中已经提到过，这是因为没有合理使用版本管理系统的分支和标签功能而产生的问题。关于使用版本管理系统有效地并行开发多个任务的方法将在第 3 章中进行讲解。

2.5.6 问题 10：在测试环境、正式环境上无法运行

这应该是开发现场常有的事，以“在自己的本地环境上能正常运行”为由，而无视 staging 环境或正式环境中发生的问题，这样的开发人员有时还是会遇到的。和这样的开发人员是无法进行沟通的。而将在开发环境中运行的内容在测试环境中运行起来要费一番功夫，将在 staging 环境中运行的内容在正式环境中运行起来也要费一番功夫，这样的话题倒也经常听说。

根据环境的不同，程序运行的动作发生变化的问题通常称为“环境依赖问题”。由于环境依赖而产生的问题究竟是怎样的呢？以下是一些常见的情况：

- 由于数据库模式的差异而产生的问题
- 没有安装程序所依赖的库而产生的问题
- httpd 或 memcached 等各种中间件由于环境不同而配置发生变化的问题

数据库模式差异的管理问题，已经在问题 4 中讨论过，具体将第 3 章中进行讲解。程序依赖库的问题也将在第 3 章的依赖关系的管理这一小节中进行讲解。

中间件等配置的问题，必然会因为环境的不同而产生差异，管理起来的确是比较困难的。在第 3 章和第 6 章中将介绍管理相关的一些小技巧。

为了实现高效、高品质的开发，这些问题不能用“环境依赖”一语带过，而应该试着去摸索解决方案。作为全世界开发人员努力的结果，

现在已经出现了帮助解决这类问题的工具。这部分内容将在第 6 章中进行讲解。

2.5.7 问题 11：发布太复杂，以至于需要发布手册

这个问题和问题 10 也是相通的。无论哪里的现场，向正式环境进行发布都是件复杂并且伴有紧张感的事情。数百行的发布手册由多人确认两三遍，即使如此谨慎地进行发布，大多数情况下也都不会很顺利。经历了各种困难终于让程序运行了起来，如果能不出故障、持续地运行的话，那就是上天保佑了。这难道不是开发现场的实际情况吗？

并且在大多数情况下，大家往往会忘记将只对正式环境进行的作业提交到代码库，或者忘记反映到发布手册上。如果还不得不去解决其他故障的话，那就更不用说了。再加上还要提交故障说明报告、向发布手册添加数量庞大的确认项目，或者确认这些项目的人数大大增加，这样一来就会使业务更加复杂。

另一方面，多数热门的 Web 服务都以惊人的势头一边修正 bug 一边开发着新功能。例如社交编程服务，也就是 Git 的托管服务——大名鼎鼎的 GitHub[①]在 1 天之内要进行 100 次以上的发布[②]。

怎样才能做到以这样惊人的速度一个接一个地进行发布呢？至少可以知道肯定不是用发布手册人工介入进行的。

就算 GitHub 这样惊人的发布频率属于例外情况，但如果部署和发布能够通过自动化简化一些的话，不仅能够减少作业中的错误，包括用户和开发人员在内，大家都会对此喜闻乐见吧。

为了实现上述内容，需要解决环境依赖的问题，还需要实现自动化测试和自动化部署。换言之就是持续地维持“随时都能发布”的状态是非常重要的。

这样的想法称为持续交付（CD）。CD 是有一定难度的实践，但一旦实现，团队的敏捷开发效率就会有飞跃般的提升。这部分内容将在第 6 章进行讲解。

---

① https://github.com/

② Deploying at GitHub（https://github.com/blog/1241-deploying-at-github）

2.6 什么是理想的项目

前几节中我们回顾了死亡行军项目的现状以及从中看出的具体问题。解决这些问题要正确执行团队开发的流程。

而正确执行开发流程则需要正确理解所使用的工具的机制，合理掌握工具的使用方法。当然不是用了工具就能解决所有的问题，其他还有团队成员的教育、思想准备等各种各样的方法论。但最重要的还是对工具的理解以及合理运用，由此也造成了软件开发的速度和质量的差异。

从第 3 章开始我们将对这些工具进行介绍，并详细讲解这些工具的使用方法、为什么需要这款工具以及应该注意哪些地方。

作为本章的总结，我们来看一下理想的项目开发流程（图 2.7）。

图 2.7　理想的开发流程示例

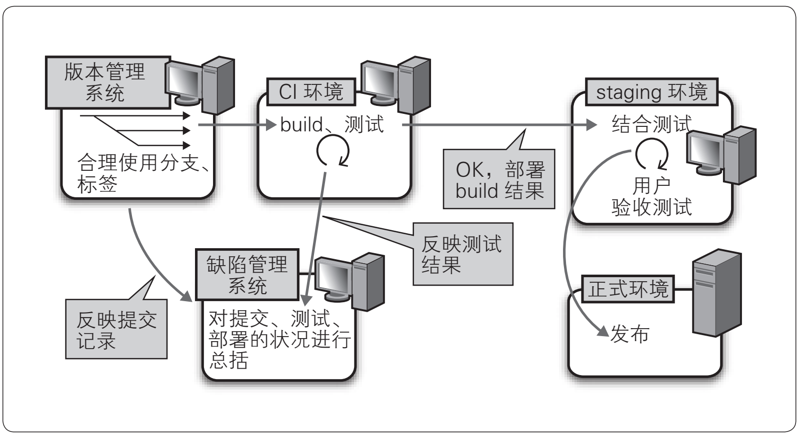


图 2.7 看起来有点复杂，但其实很简单，也就是说，只要根据本章中发生的事情以及从中反映出来的问题逆向为之就可以了。

具体可以总结为以下几点。

2.6.1　使用缺陷管理系统对课题等进行统筹管理

使用缺陷管理系统而非邮件对课题、要做的任务、发生的故障等进行统筹管理。其中，对优先度和重要度进行明确地管理，以及清楚地掌握每一个 bug 票的状态，这些是非常重要的。

易于检索，能很快地找到想找的信息，能检索到过去的 bug 票以及与其相关的处理结果，这些也是很重要的。bug 票的处理结果是版本管理中的提交记录、是 CI 系统的测试结果，需要确保能够检索到现在被部署到了哪个环境中。另外，不仅仅要向开发团队的全体成员，还要向项目的所有利益相关者[①]共享缺陷管理系统中统筹管理的信息，这点也是很重要的。

2.6.2　尽量使用版本管理系统

首先要正确使用版本管理系统，避免无意中将他人的修正覆盖。其次要合理地管理分支，明确正式环境中使用的是哪个分支，最近修正 bug 所用的是哪个分支，新功能开发使用的是哪个分支，这点很重要。这样就可以并行推进多个开发任务。

另外，通过合理地管理标签，将某个时间点发布状态的截面保存下来，这样无论多久之前的状态都能够进行回滚，这点是很重要的。还有，用版本管理系统来统筹数据库的变更管理和环境相关的配置文件的管理等也是很重要的。

2.6.3　准备可以反复验证的 CI 系统

准备好 CI 系统，时常将所有的资源集中到 1 处，并通过 build 确认是否可以正常合并，以及确认单元测试是否总能通过，这些都是很重要的。这样就能立即发现提交遗漏、修正错误等问题，从而提高软件质量。

---

① 不一定隶属于开发团队，但是会受到项目结果影响的利害关系人。

并且还需要恰当地编写测试代码和build脚本。这里的“恰当”是指无论重复执行多少次，都不会因为依赖某些状态而改变执行的结果。例如数据库状态、环境变量、中间件的配置等都适用于此。将这些内容总结在一起写成脚本，这样原本复杂的数据库构建和环境构建就能简单地进行了，执行测试的难度也会相应下降。

2.6.4 将环境的影响控制在最小限度，并随时可以发布

管理因环境不同而产生的差异的确是比较困难的，但现在有了实用的工具。如前所述，合理使用脚本和工具，对数据库的变更以及各个中间件等因环境而产生差异的内容进行版本管理。这样只要准备好可以重复执行的环境构建、发布以及动作确认的测试等，应该就可以随时发布最新的程序了。

2.6.5 保留所有记录以便日后追踪

之前已经提到了很多次，对至今为止所有的操作进行管理、记录，并做到可追踪是非常重要的。包括什么时候什么人对程序进行了怎样的修改、原来发生了怎样的问题、是否通过了测试、是否进行了发布等所有信息。

并且对上述信息进行简洁、方便的管理也是很重要的。如果用纸或Excel的工作簿，无论负责管理的人员多么努力，也无法提高开发速度。所有事情都应该实现自动化的简洁管理。

2.7 本章总结

怎么样？实现了 2.6 节中列举的关键点的项目应该可以称为理想的项目了吧？这和案例分析中死亡行军的项目有着天壤之别。可是，怎么才能做出这种理想的项目呢？

关于团队开发过程中应该采取的措施，以及各种必要的工具的使用方法，我们将从第 3 章开始进行详细讲解。

但是请不要忘记，熟练使用工具的目的是实现项目中提出的目标，也就是使顾客的价值最大化。

为了迅速并且准确地实现项目中提出的目标，应尽可能地把可以自动化的工作交给工具进行，不要在与用户价值无关的作业和没有必要烦恼的事情上浪费劳动力，这是非常重要的。这样，项目的开发人员才能将注意力集中在本来应该关注的地方，例如项目的目标是否能够达成、现在所做的是否偏离正确作法等，并最终交出完美的答卷，这才是重要的。

接下来，就让我们来学习全世界工程师们费尽心机所开发出来的实践方法和工具吧。

第3章

版本管理

3.1 版本管理系统

3.1.1　什么是版本管理系统

合理、有效地利用版本管理系统是顺利进行团队开发必不可少的、最基础的工作。是否正确理解了版本管理系统的概念及其意义将直接左右团队所发布的产品的质量中最基本的部分。

究竟什么是版本管理系统？版本管理系统是将什么时候、谁、对文件做了怎样的修改这样的信息以版本的形式保存并进行管理的系统。这里提到的文件当然包括代码，但不是说版本管理系统只能管理代码的版本。

新建表或导入数据用的 SQL 文件、构筑中间件用的配置文件，甚至是应用程序的说明手册等，只要是文件，都可以用版本管理系统进行管理。

下面我们来看一下使用版本管理系统所带来的便利之处。

3.1.2　为什么使用版本管理系统能带来便利

使用版本管理系统的优点如下所示。

- **能够保留修改内容这一最基本的记录**
- **能够方便地查看版本之间的差异**
- **能够防止错误地覆盖别人修改的代码**
- **能够还原到任意时间点的状态**
- **能够生成多个派生（分支和标签），保留当时项目状态的截面**

●……… 能够保留修改内容这一最基本的记录

什么时候、谁、对文件做了怎样的修改这些信息，虽说是最基本的，但将其作为记录保留下来也是非常重要的。在发生问题时，追踪记录（提交）能够帮助查明问题的原因。

虽然写在纸上或者使用 Excel 表格来人工进行管理也能达到同样的效果，但这样团队需要的人数就会增加。随着处理的文件种类的增加，很快这个方法就会变得不那么现实了。

最近使用上述管理方法的项目已经越来越少见了，但是在一些工程师较少的网站制作现场等，还是会有一些人工管理的地方。如果你正好在这样的现场，就算只是为了简化记录操作，也可以考虑试着使用版本管理系统。

●……… 能够方便地查看版本之间的差异

使用 `svn diff` 或 `git diff` 这样的命令就能方便地确认各个版本之间的差异[①]。可以输入命令，从命令行进行确认。还可以通过 TortoiseSVN[②]、TortoiseGit[③]、SourceTree[④]这样的 GUI（Graphical User Interface）工具进行可视化确认。除此之外，还可以使用 Trac 或 GitHub 这样的基于 Web 的代码库浏览器进行确认。

能够简单地确认版本间的差异是版本管理系统的优秀特征之一。

●……… 能够防止错误地覆盖他人修改的代码

说起版本管理系统和简单的表格管理的区别，首先想到的就是该功能。版本管理系统将文件的修改记录作为数据库进行管理，所以能够防止多人在同一时间修改同一文件的同一处。

---

① 请注意能够简单地确认差异这一点原则上仅适用于文本文件，图片这样的二进制文件则无法简单地确认差异。虽然从技术上来说是能够找出二进制文件的差异的，但是这样的差异不是人类能够理解的，只是二进制数据之间的差异。

② http://tortoisesvn.net/

③ https://code.google.com/p/tortoisegit/

④ http://www.sourcetreeapp.com/

第 2 章列举了错误地把他人的修改覆盖的例子，如果合理地使用版本管理系统，就不会发生这样的事情。

由多人修改而造成的冲突也称为 conflict 或 collision[①]。为了解决冲突的问题，版本管理系统大致提供了两类机制，分别是“锁 – 修改 – 解锁模式”（以下称为锁模式）和“复制 – 修改 – 合并模式”（以下称为合并模式）。

锁模式的做法是：在某人编辑文件期间，将文件锁住，不允许其他人对此文件进行编辑。过去的商用版本管理系统主要采用这种方式。这种方式的特点是简单，对任何人来说都是易于理解、易于操作。但是缺点在于无法多人同时进行并行开发，难以提高开发速度。

合并模式不会对文件加锁。开发者下载代码的备份进行编辑，然后再提交到代码库。提交时确认差异（diff），如果存在差异，要先对差异进行合并，然后再提交。近几年的版本管理系统基本上都采用这种方式。

CVS（Concurrent Versions System）、Subversion、Git 这些有名的版本管理系统都属于合并模式。该模式的优势在于不对文件加锁，多人可以同时获取最新的代码而不必等待他人的作业，能够并行地推进开发。和其他团队成员的修改产生冲突时，可以通过合并来消除冲突。

◆◆◆

无论是锁模式还是合并模式，合理使用版本管理系统都能够防止无意识地覆盖他人修改的代码。

专栏　锁模式和合并模式

如上所述，版本管理系统解决冲突的方式大致可分为两种，即锁模式和合并模式。

VSS（Visual Source Safe）和 Peroforce、PVCS[②] 这样的专有商用版本管理系统多采用锁模式[③]。CVS、Subversion、Git、mercurial

① 该词多用来指网络数据的冲突，所以用来指代码修改的冲突可能并不是那么的合适，但也有不少开发现场是这么称呼的。

② 专有软件（proprietary software），和它相对的是自由软件。——译者注

③ 这些产品现在也已经具备了包括合并模式在内的各种先进的功能。

等 OSS（Open Source Software，开源软件）的版本管理系统多采用合并模式。

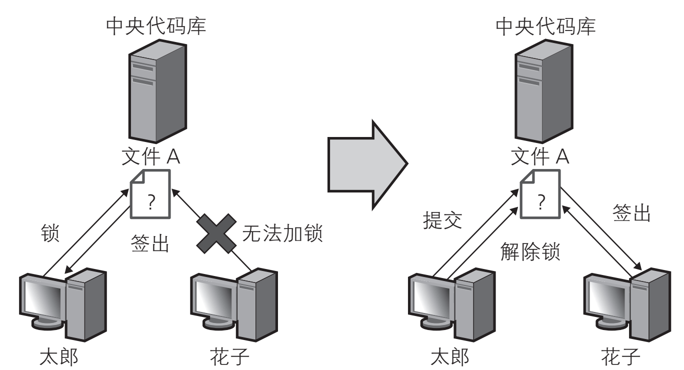
两种方法都有各自的长处和短处。近年来使用 Subversion 和 Git 的开发现场较多，合并模式也逐渐成为主流。但在大约 10 年之前，锁模式的商用版本管理系统一直都占据着主流位置。

当时开源软件还没有像今天这样被大量用于开发现场，因此比较多的是采用商用的版本管理系统。一些由那个时代的工程师主管的开发现场，至今好像依然在使用 VSS 这样的锁模式的版本管理系统。

有的开发现场虽然使用了 Subversion 或 Git，但思维方式还是基于锁模式，因此还是无法合理运用版本管理系统。

锁模式的情况下，在某人编辑文件期间，文件将被锁住，所以理论上不会发生冲突。但另一方面，多人同时并行编辑同一文件原则上也变得不可能（图 3.a），这样就大大影响了开发速度。如果有人将文件锁住后去休假了，那么开发就无法进行下去了[①]。

图 3.a　锁模式下无法同时并行地进行编辑



在太郎锁住文件 A 进行编辑期间，花子无法再对文件 A 加锁。在太郎完成编辑并解锁之前，花子只能等待

在太郎完成编辑并解锁后，花子才能将文件 A 签出。也就是说多个人不能同时编辑同一个文件

与之相对，合并模式不需要对文件加锁，所以同一文件可以同

① 实际上版本管理系统一般都有强制解除锁的功能，但这需要管理员权限，操作起来也比较麻烦。

时由多人并行地进行开发（图 3.b）。提交时会提醒你确认差异并进行合并，与他人编辑的地方发生重合时也会检测出冲突（图 3.c）。

图 3.b 合并模式下能够同时并行地进行编辑

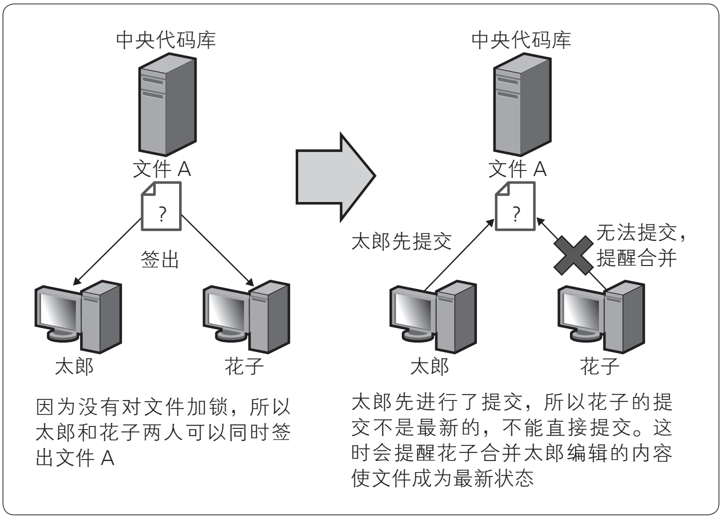
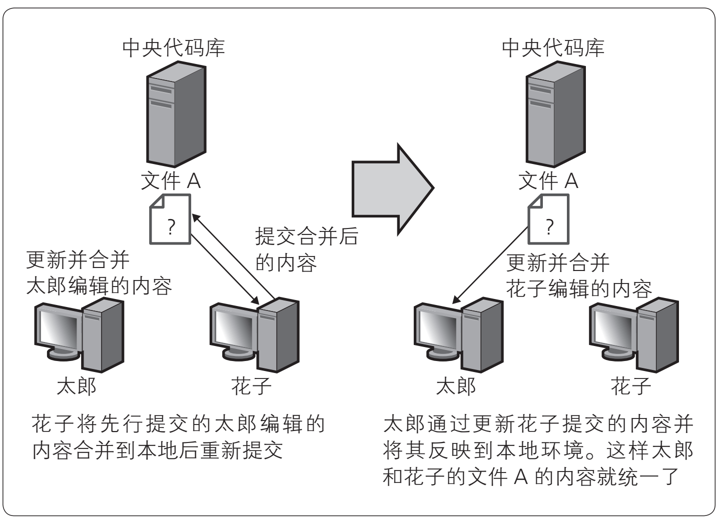


图 3.c 通过更新（update）来统一本地机器上的环境



锁模式的情况下，文件在被编辑期间是无法签出（check out）

的，所以不会发生冲突。合并模式下任何时候都可以签出文件，但随之而来的是，在你签出文件后到提交前这段时间，如果有人进行了提交，那么你就需要将这部分修改合并到本地代码后再进行提交。

这时，如果文件编辑的地方重合的话，版本管理系统会检测到冲突，并显示请手动修改冲突这样的错误消息。这是合并模式中版本管理系统的正常动作，但习惯了锁模式的人会对此感到非常奇怪。

因此，一些维护旧的开发环境的团队有时会根据锁模式和合并模式的这些差异，固执地认为 Subversion 和 Git 无法锁住文件，从而造成冲突频发，无法有效地管理代码。这样的现象在一些习惯了锁模式的开发现场尤为显著。

实则恰恰相反，锁模式的版本管理系统由于效率较低，无法合理地进行版本管理的情况较多。

锁模式的版本管理系统在文件加锁的情况下拒绝其他人员对此文件进行编辑，这样的确不会造成冲突。但实际开发中往往不允许这样“慢条斯理”，于是开发人员就会无视文件被锁住，独自在本地进行开发，等待锁解除后再手动合并并提交。

各个版本管理系统可能有所差异，但多数采用锁模式的产品都没有自动检查差异并进行合并或者检测冲突等功能，即使有也非常弱，因此容易发生手动合并时不小心将他人的修改覆盖的情况。并且加锁也不能说是绝对的，也有将锁强制解除并覆盖提交的功能。

举一个笔者亲眼所见的例子：一位习惯了锁模式的开发人员在使用 Subversion 这样的系统时，因为讨厌发生冲突，所以没有使用 svn update[①]，而是每次都将代码下载到开发目录以外的目录中，再复制自己编辑过的代码替换原有代码后进行提交。这种做法真是令人吃惊又哭笑不得。

这样的做法的确不会发生冲突，但随之而来的是将频繁发生他人的修改被覆盖的事情。实际上那个开发现场发生的 bug 和退化实在太多，让人觉得项目已经处于崩溃的边缘。而且笔者还记得当时他还被视为现场比较有经验的开发人员，所以事态就更为复杂了。

如果大家的工作单位现在还在使用锁模式的版本管理系统，或

① 从代码库中取得最新的代码合并到本地的命令。Git 的情况下是 `git pull origin master`。

者像上面那样虽然使用了合并模式的系统但使用方法有误的话，那么您可以先和周围的同事聊一下本专栏的话题，并试着劝说他们改变用法。

但是也有例外，例如在管理图像这种二进制文件的情况下，因为它和代码这样的文本文件完全不同，无法进行合并，所以用加锁的方法效率往往会高一些。此外，在制作用于重要发布的包时，如果作业时间长达数小时，有时就会特意加锁以保证这段时间内绝对没有其他人员的修正加进来。虽然近年来合并模式已经成为主流，但大多数的版本管理系统对于锁模式也是支持的。

使用版本管理系统时，理解锁模式和合并模式这两种版本管理系统的差异并合理使用，这才是最重要的。

●……能够还原到任意时间点的状态

因为保存着过去的修改记录，所以在发生任何问题的情况下，例如在发生退化时或者新添加的功能不再需要时，理所当然地能够立即回退到过去任意时间点（的版本）。

不同的版本管理系统对于版本的思考方式也有所不同。从历史上来说也大致可分为两类：基于文件和基于变更集（changeset）。

例如 CVS 是基于文件的版本管理，即对每一个文件分别进行版本管理。与之相对应，Subversion 及之后（包括 Git）的主要的版本管理系统都是基于变更集的。变更集将对多个文件的一次修改看作是理论上的一个单位。基于变更集的版本管理系统就是以此单位来分配版本号的。

基于文件的情况下，如果要取得过去某个时间点的版本，就需要集齐每一个文件所对应的正确的版本，例如文件 A 是 1.1 版、文件 B 是 1.9 版、文件 C 是 2.3 版。与之相对应，基于变更集的情况下，因为是将修改合并后进行管理的，所以要取得过去某个时间点的版本时，只需要知道其版本号就能够完整地获取整个项目。

如上所述，不同的版本管理系统对于版本管理的思考方式虽然有所不同，但都可以随时回退到过去的任意时间点。这是使用版本管理系统的一个很大的优势。

专栏 基于文件和基于变更集

版本管理的思考方式从历史上来看有基于文件和基于变更集这两种。

VSS（Visual Source Safe）这样历史悠久的商用产品，以及开源软件中 CVS 这样较老的工具，都是以基于文件的方式来实现的。

那些太习惯于使用 VSS 和 CVS 而不熟悉 Subversion 及其以后的工具的开发人员，可能是因为他们以基于文件的方式来理解版本管理，所以往往会胡乱地把提交的粒度分得很细——将文件一个一个地进行提交。因为 VSS 和 CVS 是基于文件的方式来管理版本的，所以逐个提交文件和打包一起提交从结果上来看是完全一样的。甚至可以说在 VSS 和 CVS 的情况下，将文件一个个地分别提交更为直观。

而 Subversion 及其以后的版本管理系统则以变更集为单位进行版本管理。因此如果将文件一个个地分开提交，变更集也会被分为多个，这样一来，变更集原本具有的能够为解决某个问题而进行修正的意义就完全丧失了。

版本管理系统进化为能够对变更集进行管理的意义在于让每一个版本都有自己的含义。因此含义相同的修改就应该置于同一个变更集中一起提交。

如果开发团队中有成员一直使用 VSS 或 CVS 等工具，并且对其他的版本管理系统一无所知的话，可以向他介绍本专栏的内容，并试着让他重新认识版本管理系统的使用方法。

●……能够生成多个派生（分支和标签），保留当时项目状态的断面

版本管理系统有分支管理和标签管理的功能。

第 2 章的案例分析中列举了无法高效地在新功能开发和已发布版本的 bug 修正之间切换的问题[①]，其实只要合理地进行分支管理，就能够解决这样的问题。通过使用分支管理功能，项目就可以在多个方向上建立分支，例如可以分别建立新功能开发分支和已发布版本的分支。

① 请参考 2.2 节。

各个版本管理系统在实现的细节上可能有所差异，但通常都具备对分支进行合并的功能。这个功能越是完善，就越有勇气去挑战困难的开发。

不同的版本管理系统在分支切换、合并的速度以及正确性方面是有差距的，近年来 Git 是在分支管理方面做得最好的。

标签管理是能够对文件或者变更集任意命名（打上标签）的功能。利用这个功能可以对过去任意时间点的系统快照进行管理。

可以像阿尔法版、贝塔版、发布版这样，或者像版本 0.1、0.2 这样根据每个产品的外部版本号[①]来打上标签，一般都是在到达某节点时为项目打上标签的。不同版本管理系统的标签功能在细微的动作或规格上可能有所差异，但思考方式大致是相同的。

① 这里并不是指版本管理系统内部的版本号。

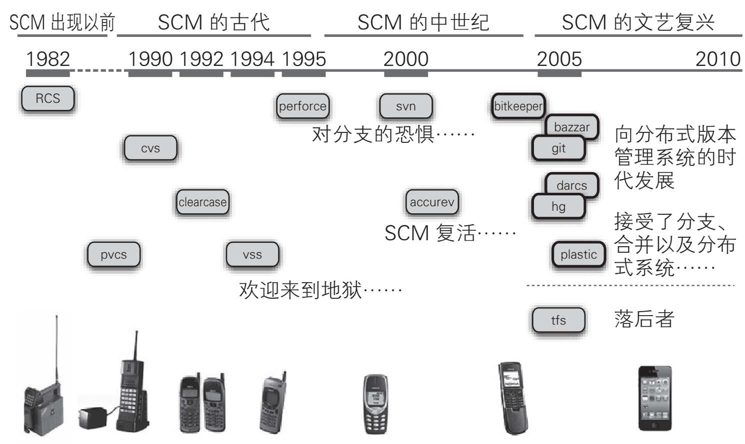
3.2 版本管理系统的发展变迁

接着我们将按时间顺序来介绍具有代表性的版本管理系统。

版本管理系统是高效软件开发中必不可少的最基本的要素。因此在软件开发的历史上出现过很多版本管理系统的概念，这些概念也得到了实现。

这里让我们来简单回顾一下版本管理系统的历史，看一下与时俱进的版本管理系统的思考方式（图 3.1），同时也了解一下版本管理系统是如何发展到现在被认为是最有效率的分布式版本管理系统的。

图 3.1 版本管理系统的历史



※SCM 是 Source Code Management（代码管理系统）的简称。
本图参考了“The version control timeline”by The plasticscm blog
（http://codicesoftware.blogspot.com/2010/11/version-control-timeline.html）。

3.2.1 没有版本管理系统的时代（20 世纪 70 年代以前）

20 世纪 60 年代～20 世纪 70 年代可以称为“版本管理系统的史前时代”。那时候笔者还没有出生。那个时代还没有可以称为版本管理系统的产品。

当时可能是以日期命名的目录来管理，或者制作表格文件来管理。有些开发团队也有可能在内部秘密地开发类似于版本管理系统的产品并独自使用。现在已经无从考证了。

3.2.2 RCS 的时代（20 世纪 80 年代）

现在某些 UNIX 操作系统上仍然捆绑安装的 RCS（Revision Control System）是于 1982 年发布的。它以管理本地机器上的文件为目的，是最早的真正意义上的版本管理系统。

RCS 具备了对文件之间的差异进行管理这种基本的版本管理机制，但还没有考虑到多人开发的情况，原因是要将管理的版本和多个人进行共享是非常困难的[①]。因此，这只是一款在本地机器上对文件进行管理的产品。

3.2.3 CVS 的诞生（20 世纪 90 年代）

RCS 之后也发布过一些版本管理系统，但都是只能在本地机器上对文件进行管理，对多人开发的支持不够这一功能上的缺陷一直没有得到改善。

改变上述状况的是 1990 年左右发布的 CVS（Concurrent Versions System）。通过 CVS 名字中的 Concurrent（并行）就能知道，这是一款为了支持多人并行开发而设计的系统。

和 RCS 不同，CVS 采用了客户端 / 服务器的架构，通过 CVS 服务

① 当然也并非完全不可能，通过共享目录，在 rcs 文件中粘贴符号链接，还是可以实现共享的。那时候一般所有的开发都在一台机器上进行。

器使多人共享代码成为可能。从服务器签出代码进行编辑，然后再提交到服务器，这种现在看来理所应当的作业流程，可以说是由 CVS 最先确立的。CVS 还采用了合并模式来解决冲突，使得多人并行编辑同一文件变得容易进行。

但另一方面，版本管理仍采用基于文件的方式实现，每一个文件都有自己独立的版本号。因为没有变更集的概念，所以发生 bug 时要找出相关的一系列文件及其版本号是一件比较困难的事情。

虽然 CVS 实现了分支管理和标签管理，但创建分支是重量级的作业。特别是将分支再度合并回主干时的作业异常困难，根据项目的规模，有时不得不任命专门的人员来进行此作业。

CVS 是最早考虑到使用网络的版本管理系统，并且属于开源软件，使用是免费的，所以从 20 世纪 90 年代开始就被广泛使用。在一些开发现场，至今仍然在使用 CVS。

3.2.4 VSS、Perforce 等商用工具的诞生（20 世纪 90 年代）

在 CVS 被发布的几乎同一时期，各个软件企业也发布了多款商用的工具。具有代表性的是 PVCS、ClearCase、VSS 和 Perforce 等。

很多历史较久的开发团队至今还使用着上述版本管理系统。它们的后继产品如今多是将项目管理等功能整合起来，以 ALM（Application Lifecycle Management）工具的形式出售。

这些商用工具和 CVS 相同，也采用客户端 / 服务器模式。各个产品在实现的细节上有所差异，但大多都是以锁模式来消除冲突，并基于文件进行版本管理。

当然，这是 20 世纪 90 年代当时的情况，这些工具的最新版本改进并添加了各类功能，都有着易于使用的特性。

20 世纪 90 年代出现的这些商用工具各具特色，被市场广泛接受，但它们都没能解决分支管理和合并困难的问题。

3.2.5 Subversion 的诞生（2000 年以后）

进入 2000 年后，作为 CVS 的后继，Subversion 诞生了。它和 CVS 一样采用了客户端 / 服务器模式，通过合并模式来消除冲突。同时还引入了变更集的概念，使得赋予相关联的文件相同的版本号成为可能。据此，bug 的调查也变得容易不少。

并且 Subversion 还可以对 CVS 不支持的文件名和目录名的修改和删除操作进行追踪。对项目的管理也更为灵活，这也是 Subversion 的特色。

Subversion 可以说是现在最普及的版本管理系统。特别是它的 GUI 客户端工具 TortoiseSVN 非常完善，被 Windows 操作系统的用户广泛接受。

虽然分支的创建已经变得简单且快速，但是将分支合并回主干的作业依然非常费劲。因此多数开发现场都尽量避免使用合并，把分支的建立控制在最小限度。

3.2.6 分布式版本管理系统的诞生（2005 年以后）

上面介绍的 CVS、Subversion 以及各个商用的工具都是基于客户端 / 服务器模式来实现的。代码库只存在于服务器上，各开发人员从服务器上下载代码，编辑后再提交到服务器。

2005 年出现的 Git 改变了上述方式。

Git 的开发者是著名的 Linux 之父 Linus Torvalds。Linux 社区原本使用的是专有[①]的分布式版本管理系统（Distributed Version Control System，DVCS）BitKeeper[②]，但后来因为某些原因不得不放弃使用 BitKeeper，于是就开始了 Git[③]的开发。

Git 出现之后，开源项目业界使用分布式版本管理系统的项目逐渐

① 由特定的软件生产商发布的规格等不公开的产品。

② http://www.bitkeeper.com/

③ 放弃 BitKeeper 的原因比较复杂，其中涉及了商业协议等。有兴趣的读者可以参考 http://www.path8.net/tn/archives/6039。——译者注

增加，分布式版本管理系统逐渐扎根下来。如今已经普及到了开源项目以外的一般项目中。分布式版本管理系统中除了最有名的 Git 之外，其他还有 Mercurial[①]、Bazaar[②]等。

分布式版本管理系统最显著的特征就是没有中央代码库，完全采用 peer to peer 的模式。开发人员并不是从中央代码库中下载代码，而是将代码库完整地克隆到本地机器上，使得本地机器上保存有完整的代码库备份[③]。据此，不需要借助网络就能够完成所有的操作，因此具有提交等命令执行速度快的特征。因为向本地代码库提交的速度很快，所以可以频繁地进行提交，以降低工作成果丢失的风险。

分支也是一样，由于只需自己本地的代码库就能建立分支，因此在想进行某种尝试时，就可以随意地建立分支。这和 Subversion 之前的只能在中央服务器上建立分支的中央集权型版本管理系统有很大的区别。

当然，Subversion 之前的产品中已经实现的变更集的管理和合并模式的冲突消解等功能，Git 都继承了下来。

总之，提交、分支、合并的低开销，只要正确提交就几乎不会丢失作业，试验性质的分支建立方便，能够实现灵活的工作流程，这些都是 Git 比较主要的特征。分布式版本管理系统的出现，可以说是版本管理系统和团队开发的重大进步。

但是难点在于学习成本。由于分布式版本管理系统（在系统上）没有中央代码库的概念，和之前的版本管理系统的思想方式有着根本性的差异，因此初次使用的人基本都会感到不知所措。

但分布式版本管理系统是值得花费这样的成本的。世界上几乎所有的开源项目都采用了分布式版本管理系统，从这点就能看出。

3.2.7　番外篇：GitHub 的诞生

虽然有些偏离版本管理系统的历史，但这里我们还是要提下 GitHub。

---

① http://mercurial.selenic.com/

② http://bazaar.canonical.com/

③ 虽说机制上不需要中央代码库，但实际使用中往往还是会设立中央代码库。

GitHub 是 2008 年诞生的 Git 项目的在线托管服务。“社交编程”这个标语可能更为出名[①]。

GitHub 之前也有代码托管服务，如 SourceForge[②]和 assembla[③]等。这些服务的特点是能够方便地托管代码，还支持多人开发和缺陷管理，因此主要在开源软件界被广泛使用。

GitHub 是上述服务的 Git 版，和上述托管服务相比，GitHub 更为流行和普及。

GitHub 流行起来的原因有很多，其中最主要的是得益于 Fork 和 Pull Request 这两个特色功能。如果别人的项目中有你感兴趣的地方，可以方便地克隆该项目（Fork）并添加修改，再发送补丁申请将自己的修改反映到原来的项目中（Pull Request）。

之前的中央集权型版本管理系统是无法实现上述功能的。首先是无法对已有项目进行克隆。另外，发送给项目的补丁（修改）如果不能立即反映到项目中，也得不到保存，最终就会丢失。

GitHub 的成功之处可以说在于将分布式管理系统（Git）“不存在中央代码库”这一特征所具有的潜能最大限度地激发了出来。原本以 Git 为代表的分布式版本管理系统有着运行速度快、建立分支简单快捷、合并方便等特点，但同时也存在 API 比较复杂，不了熟悉的话很难灵活应用这样的问题。

GitHub 可以说是通过易用易懂的 UI 和高超的设计灵感将 Git 的潜力充分激发了出来，一下子改变了开发人员的世界。

如今参加开源项目的门坎已经大大降低。如果开源代码中有什么看不惯的地方，或者发现了 bug，都可以 Fork 并进行修改。

修改顺利的话，可以向原来的项目进行 Pull Request，将修改反映到项目中。这样就对开源项目做出了自己的贡献，不需要其他任何杂七杂八的事情。

如此便捷的 GitHub，如果只用在开源软件界就太浪费了。事实上，

① 到 2014 年，GitHub 网站上社交编程这个标语已经没有了。

② http://sourceforge.net/

③ https://www.assembla.com/

世界上的很多公司都已经在使用 GitHub 开发自己的产品。可以毫不夸张地说日本国内主要的互联网服务提供商几乎都在使用 GitHub。

笔者现在所属的公司也在使用 GitHub。一旦了解了 GitHub 的便利性和分布式版本管理系统的自由性，就很难再回到之前的中央集权型版本管理系统了。听说一些资深的工程师，换工作时都会确认对方企业是否使用 GitHub。

GitHub 的共享代码库是免费的，但私有的代码库还是收费的，这点请注意。并且 GitHub 还有供公司内部局域网使用的产品 GitHub Enterprise①。企业在开发产品时使用私有代码库的花费②或者 GitHub Enterprise 的花费需要预先纳入到项目预算中。但这样的花费还算是物有所值的。③④⑤

3.2.8 版本管理系统的导入情况

走马观花地看了一遍版本管理系统的历史。从没有版本管理系统到 RCS 的出现，CVS、Subversion 的出现，再到分布式版本管理系统的 Git 的出现，仅仅用了 20 年的时间。在这期间，版本管理系统从基于文件到基于变更集、从锁模式到合并模式、从中央集权型版本管理系统到分布式版本管理系统，有了巨大的进步。这些进步可以说都是人们为了使多人同时并行开发更容易，以及提高开发速度而努力的结果。

取得这样的进步只用了 20 年的时间，可以说是相当迅速了。当时

① https://enterprise.github.com/

② https://github.com/pricing

③ 根据代码库和用户数量的不同，有各种价格的套餐。具体请参考 GitHub 的网站。

④ 也有和 GitHub 功能几乎完全相同，并且私有代码库在一定程度上也可以免费使用的服务，那就是 Bitbucket（https://bitbucket.org）。Bitbucket 原本是 Mercurial 这款分布式版本管理系统的托管服务，现在也支持 Git 了。在公司内部导入 GitHub 之前，可以先使用 Bitbucket 的私有代码库，让开发人员先体验一下分布式版本管理系统和 Pull Request，之后再正式地导入 GitHub。或者就这样继续使用 Bitbucket 也是一个不坏的选择。

⑤ 在某些情况下，可能会由于安全策略的问题而不允许使用 GitHub 这样的外部服务，或者没有使用 GitHub Enterprise 的预算。这时可以选择使用 Gitlab（http://gitlab.org）、GitBucket（https://github.com/takezoe/gitbucket）等 GitHub 的克隆。

刚刚开始工作的 20 岁左右的工程师，如今已 40 岁左右，并成为了开发现场的中坚力量。到了 40 多岁这个年纪，很多工程师都已经成为项目经理，或者对工具和技术的选择拥有决定权。当然也有人依然在开发现场编写代码。

特别是在日本，技术方面的信息传入比起美国要晚一些。20 世纪 90 年代初出现的 CVS 好像到 90 年代后期才开始被广泛使用，2000 年出现的 Subversion 到 2004～2005 年才逐渐普及。至于 2005 年出现的 Git，到 2010 年才有一部分先进的公司开始使用。直到 2014 年，多数互联网公司才开始讨论采用 Git。这些都是事实。

如今还在使用 20 世纪 90 年代发布的商用版本管理系统的公司多数是企业级的系统集成商（SIer）。特别是日本企业，它们有着只要系统尚可使用就不会强行升级这样独特的文化，状况就可想而知了。

你所在的公司在使用 Git 或者 GitHub 吗？如果是的话那么你是幸运的。你所在的是非常理解技术的优秀的公司，或者是最近成立的创新公司。请珍惜这样的环境。

如此幸运的人应该不会太多。比如有些开发现场的项目经理当年在一线工作时只能采用 CVS 这样基于文件的版本管理，或者用惯了 VSS 这样基于锁的难以并行开发的系统，这样的开发现场往往不会使用 Git 或 GitHub。另外，由于企业文化保守，至今仍在使用 VSS 的企业，以及只是形式上导入了 Subversion，用法和 VSS 的时候基本没有区别的企业也是比较多见的。

再重复一下，版本管理系统只有短短 20 年的历史①。

假如你的上司、你所在的公司对于版本管理的理解不足，还在使用旧的模式进行管理，这是对技术方面的懈怠而造成的结果。在认识到这一点的基础上，请从自己开始努力，来改变这样的公司文化。

① 不只是版本管理系统，本书中出现的工具和插件都在这不到 20 年的时间内有了巨大的进步。

3.3 分布式版本管理系统

3.3.1　使用分布式版本管理系统的 5 大原因

2014 年最先进的版本管理系统——分布式版本管理系统的优点有以下 5 个。

●能将代码库完整地复制到本地

这是和中央集权型版本管理系统的主要差异。之后列举的优点几乎都是基于这个特点的。

●运行速度快

本地机器上存放有所有的文件，因此通信开销低，提交、分支、合并等操作速度快，这是分布式版本管理系统的一大优势。

●临时作业的提交易于管理

和中央集权型版本管理系统不同，分布式版本管理系统可以在不影响全体代码库的情况下在本地提交代码。这样可以方便地保存临时作业，有助提高开发效率。

另一个优点是可以不用考虑对整体的影响，方便地进行实验性质的开发。

●分支、合并简单方便

相同的内容已经提到过很多次了，分支建立简单、快速，不仅能提高开发速度，也不会阻碍你的各种尝试，这也是一大优点。

●……… 可以不受地点的限制进行协作开发

借助本地环境中保存有全部文件这一特性，无论在无法连接互联网的离线环境，例如在乘坐飞机时，还是访问互联网比较困难的偏远地区，都能够顺利地进行开发。可以说最适合不限地区和国家的开源代码项目了。并且该特性最近还起到了将软件开发企业从地域限制中解放出来的作用，以美国西海岸的企业为中心，开发人员的多国籍化、多地区化正在推进之中。

3.3.2 分布式版本管理系统的缺点

分布式版本管理系统作为当前最先进的版本管理系统，基本上是优点占大部分。但缺点或者说需要注意的地方也并不是没有。

●……… 系统中没有真正意义上的最新版本

因为系统中没有中央数据库，所以也不存在最新版本这一概念。Subversion 的话，最新版本自然就是 trunk 上的 HEAD。而分布式版本管理系统的情况下，如果没有在运用策略上确定中央数据库，那么最新版本也就无从谈起。

●……… 没有真正意义上的版本号

没有中央代码库就意味着不知道哪个是最新版本，或者说哪个都有可能是最新版本。也就是说，即使像 Subversion 那样连续地分配版本号 1、2、3 也没有任何意义。例如版本号 231，至于代码库 A 的 231 版本和代码库 A' 的 231 版本哪个是新版本，就完全不知道了。

因此，分布式版管理系统为每一个变更集分配了 GUID[①]（Globally Unique IDentifier）。也就是说，并不是管理版本的新旧，而是管理变更集的唯一性。这样的管理方法与合并的便利性之间有着密切的关联，但

① 是指全局唯一标识符。原本是不重复的 128 位的整数，考虑到可读性方面的问题，分布式版本管理系统在使用时采用了“26fde9ed06bc4d0f8773cb399e73eb63”这样的 32 位十六进制的表现形式。

开发人员之间的对话就比较麻烦了。这是因为必须用隐晦的 GUID 来称呼修改。比如“231 版本”，就不得不改称为“‘26fde9ed06bc4d0f8773cb399e73eb63’的修改”，如果没有习惯的话，还真是让人吃不消。

●……… 工作流程的配置过于灵活，容易产生混乱

分布式版本管理系统中不存在中央代码库，任何一个代码库都是等价、平行的。系统上可以从任意一个代码库向另一个代码库自由地通过 Push/Pull 来发送修改补丁。这样的形式能够灵活地组建工作流程，可以算得上是优点。但如果团队成员没有习惯的话，反而容易产生混乱。

因此，实际运用中会设立中央代码库，各开发人员从中央代码库克隆代码进行开发，这是比较常规的做法。如此一来，实际运用中使用 GitHub 就成了最方便的选择，这样做的开发团队也比较多见。

●……… 思维方式的习惯需要一定的时间

如上所述，分布式版本管理系统的缺点的根本原因在于：在使用分布式版本管理系统时需要在至今为止的中央集权型版本管理系统的基础上进行范式转变①。

分布式版本管理系统和 CVS、Subversion、VSS、Perforce 的构造都不类似，所以越是习惯以前的版本管理系统的人，在一开始时就越是容易感到困惑，操作也容易失败。这也是导入分布式版本管理系统最大的壁垒，好在现在已经出现了较多的入门书籍可供参考。

① 即思维方式的根本转变。——译者注

3.4 如何使用版本管理系统

从这里开始我们一起来看一下为了顺利地推进团队开发，我们应该用版本管理系统具体管理些什么，以及如何管理。

3.4.1 前提

本节的内容在没有特殊说明的情况下，都是使用 Git 和 GitHub 来进行说明的。但这里不会对 Git 的安装方法和基本命令等的使用方法进行说明。另外，虽然文中会出现一些 Git 命令，但也仅仅展示了这些命令的一般用法，这点请注意[①]。

Git 的使用门槛虽然有些高，但它非常灵活易用，并且几乎包罗了所有的需求和用例。因此当你觉得 Git 无法实现某些功能时，可以查看一下相关网页[②]或参考书籍，还有 `man`[③]，一定能找到实现的方法。

3.4.2 版本管理系统管理的对象

应该用版本管理系统进行管理的对象可谓是多种多样，用一句话来概括就是“能够管理的对象都应该用版本管理系统进行管理”。

根据《程序员修炼之道：从小工到专家》[④]一书中的解释，其理由如下。

---

① Git 拥有非常灵活的命令体系，执行相同的内容可以有多种做法。

② 可以参考如下资料。

・猴子都能懂的 GIT 入门

http://backlogtool.com/git-guide/cn/

・*Pro Git*

http://git.oschina.net/progit/

③ 在 UNIX/Linux/MacOSX 环境下执行 `man git` 命令就会显示 Git 的帮助手册。

④ Andrew Hunt、David Thomas 著，马维达译，电子工业出版社，2011 年 1 月。

——译者注

把项目整体纳入到代码管理系统之中，这样的做法隐含着显著的优点。那就是能够实现输出文件的 build 作业自动化，并且能够反复执行。

> 重要的是通过使用版本管理系统来管理所有项目发布所需要的文件，这样就可以实现 build 的自动化。如果为了生成产品发布所需要的文件，每次都要委托 QA 部门[①]进行测试，或者委托运维部门手动生成文件，那是不可能实现优质快速的开发的。

上述思考方式称为持续集成（Continuous Integration，CI）和持续交付（Continuous Delivery，CD）。CI 和 CD 在这几年已经成为了热门词汇，其实在很久之前已经有一部分开发团队对此进行了实践。二者相关的内容将分别在第 5 章和第 6 章进行说明。

为了实现 CI 和 CD 这样的实践，首要前提就是使用版本管理系统对所有必要的信息进行适当的管理。一般我们所说的应该管理的信息主要包括以下这些。

- **代码**
- **需求定义、设计资料等文档**
- **数据库模式、数据**
- **中间件等的配置文件**
- **库的依赖关系定义**

●……… 代码

版本管理系统最基本的功能就是对代码进行管理。从版本管理系统也称为代码管理系统（Source Code Management，SCM）就可以看出，版本管理系统原本就是为了管理代码而设计的。

优秀的团队开发，首先就要从对代码进行版本管理开始做起。这里所说的代码也包括测试代码。测试代码的写法将在第 5 章中进行讲解，对测试代码也要进行版本管理。

① 负责确保软件质量的部门。

●········ 需求资料、设计资料等文档

对项目相关的文档也应该尽可能地进行版本管理。根据所属团队和项目的不同，有些文档是由销售人员或者企划人员撰写的，有些是由程序员自己写的，可谓是多种多样。文档一般反映了项目的起因，即为了解决什么问题而开展的，是为了和后继者交接工作内容所写的非常重要的资料。并且随着项目的进行，对文档进行相应的修改也是常有的事情，因此要尽量一同进行版本管理。

如果可能的话，请尽量用文本文件的形式来管理这些文档。现在已经有了 Markdown[①]、Textile[②]、reStructuredText[③]等多种格式，它们虽然是文本形式，但在一定程度上也能制作出漂亮的文档。

●········ 数据库模式、数据

现在可以说几乎不存在不使用数据库的 Web 应用程序。这也意味着没有理由不对数据库模式和数据[④]进行版本管理。

正如我们在第 2 章的案例中所见到的那样，是否能够管理好数据库模式及数据将直接决定团队开发是否能够顺利进行。本章稍后会介绍成功管理数据库的机制。

●········ 配置文件

应用程序一般都会使用到一些 Web 应用程序框架以及中间件。这些 Web 应用程序的框架和中间件的配置文件也应该进行版本管理。

根据部署环境的不同，配置文件中的值也会相应地改变，因此就必须根据环境把配置文件分开来。Ruby on Rails 和 Play Framework 等最近的 Web 应用程序框架提供了针对不同环境进行配置的机制，可以直接使用。

---

① http://daringfireball.net/projects/markdown/

② http://textile.sitemonks.com/

③ http://docutils.sourceforge.net/rst.html

④ 这里的数据不是指随着用户的操作等而增加的事务数据，而是指初始化设置这样的程序启动所必需的主数据（master data）。

但是，Apache、PostgreSQL、memcached 等应用程序的配置文件大都没有提供根据环境进行配置的机制，这方面只能自己花一些工夫了。

●……… 库的依赖关系定义

Java 的话有 Apache Commons[①]系列的库，脚本语言的话有 ImageMagick[②]和 Mechanize[③]等，一般开发过程中都会用到一些外部的库。这些外部库的依赖关系也应该纳入到版本管理系统中进行适当的管理。

① http://commons.apache.org/

② http://www.imagemagick.org/

③ http://mechanize.rubyforge.org/

3.5 使用 Git 顺利地推进并行开发

这里我们来看一下通过使用版本管理系统，在团队中顺利地开展并行开发的方法。

如上所述，Git 提供了能够使多人并行开发的合并模式，因此能方便地实现多人同时编辑同一文件。但是，当一个人不得不并行地进行多个不同的开发时，就会有一些问题。

第 2 章的问题 2 和问题 9 中涉及的、在新功能的开发过程中需要修改已发布版本中的 bug 的情况下，就可以使用版本管理系统的分支功能。

3.5.1 分支的用法

●……… 什么是分支

分支（branch），英语的意思为树枝。是一种从主干分离出来的，作为和主开发内容不同的开发内容进行分别管理的机制。分支是几乎所有的版本管理系统都具备的标准功能。

和其他的版本管理系统相比，Git 的分支功能的特点是速度快且易于使用。

●……… 什么是发布分支（release branch）

以第 2 章的情形为例，我们来看一下分支的使用方法。但在此之前，我们先来谈一下发布分支的用法。

发布分支是向正式环境进行发布时实际使用的分支。有将 master 用作发布分支的，也有从 master 建立别的分支作为发布分支的。

具体来说就是要根据各团队和项目的实际情况选择合适的方法，没

有一定正确的答案。关于使用 Git 的工作流程的组建方式，我们将稍后介绍。本次事例中暂且以 master 作为发布分支，下面的讲解都是以此为前提的。

●……… 克隆和建立分支

首先将（所认为的）中央代码库克隆到本地机器上。

```
$ git clone git@github.com:CoolInc/someniceapp.git
```

接着为发布后的新功能开发建立分支。分支的名称就是 new-cool-function。

```
$ git branch new-cool-function
```

分支建立之后进行 checkout 操作。这里的 checkout 和以 Subversion 为代表的中央集权型版本管理系统中的 checkout 的意思是不同的，这里使用 checkout命令来进行分支切换。

```
$ git checkout new-cool-function
```

这样一来就可以在 master 以外的分支上进行新功能的开发，不会对已经发布的代码产生任何影响。

●……… 提交和提交记录

假设已经修改了几处代码，现在我们用 git commit命令来试着进行提交。

```
$ git commit -m "实现了非常炫的功能"
$ git commit -m "修改了一些小错误"
$ git commit -m "想到了一些需要改进的地方，于是进行了修改"
```

上述提交记录的问题在于内容过于抽象[①]，但我们这里暂且先继续下去。

再用 git log命令来确认一下提交记录。

---

① 阅读到这里的读者一定知道这种写法的提交记录是肯定不行的。这里我们是以第 2 章的案例进行说明，第 2 章的案例中也正是因为没有正确地填写提交记录，才导致提交记录如同上述提交记录一样让人无法理解。实际操作中应该将修改的内容写入提交记录。关于这部分内容我们在第 4 章会进行具体的讲解。另外，在提交记录中记下相关联的 bug 票号也是非常重要的。

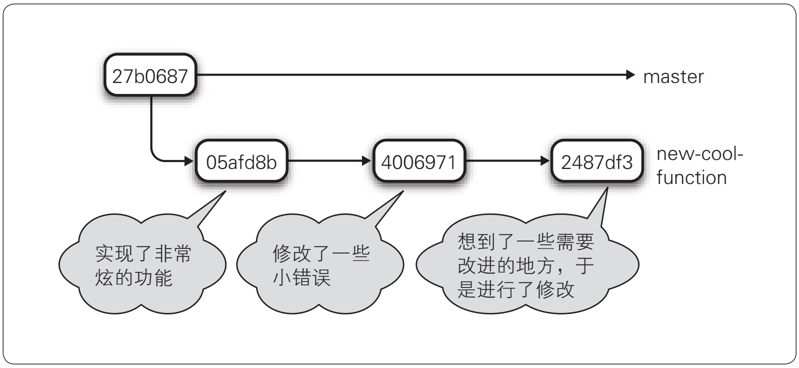
```
$ git log --pretty=oneline
05afd8b298d2439ddf7d5ae720b4967613fb11cb 实现了非常炫的功能
4006971346b0cae1596914023981eff4a8b5410c 修改了一些小错误
2487df32b6096cd349d8543304a299e41fdb037a 想到了一些需要改进的地方，于是
                                         进行了修改
```

还可以用 `git branch` 命令来确认分支的改变。

```
$ git branch
  master
* new-cool-function
```

至此，新建分支的状态如图 3.2 所示。

图 3.2 新建的分支



●……… 分支的切换

在第 2 章的案例中，有报告称已经发布的正式环境中发生了故障。如果按照第 2 章的死亡行军项目的做法，又是给目录重命名又是复制文件的话，那就实在太麻烦了。其实用 `git checkout` 命令就能简单处理。

```
$ git checkout master
```

只需执行上述命令就能切换回 master 分支，即本次事例中的发布分支。如果不确定是否已经回到 master 分支，可以通过 `git log` 命令确认下提交记录。

```
$ git log --pretty=oneline
27b06870957e44d5b02606af668c9a120df4b7e0 最初的发布版本
```

保险起见，我们再用git branch命令来确认下。

```
$ git branch
* master
  new-cool-function
```

确实已经回到了 master 分支。这里假设我们正处于第 2 章所述的境况下，让我们试着对故障进行处理。不直接修改 master，而是新建故障处理用的分支，然后再进行提交。Git 的分支建立非常快速且易于合并，因此应该灵活应用其分支功能。

```
$ git checkout -b issue345
$ git branch
* issue345
  master
  new-cool-function
```

可以使用git checkout -b XXX命令来建立分支并同时进行 checkout 操作。分支的名字可以任意取，这次假设已经有了故障报告的 bug 票（ticket），以票号为名字建立分支。缺陷（ticket）管理系统（ITS/BTS）的相关内容将在第 4 章进行讲解。

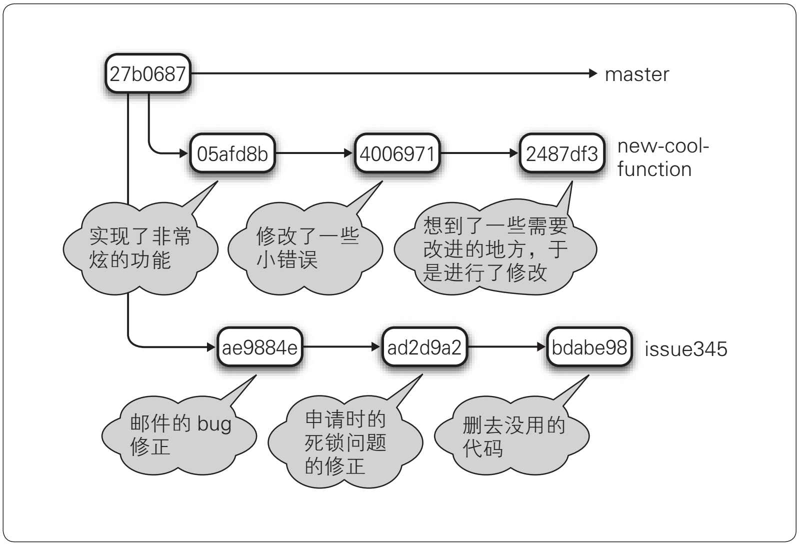
●……… 修正 bug 后的提交

在 issue345 分支上对 bug 进行了修正，并进行了 3 次提交，如下所示。

```
$ git commit -m "邮件的bug修正"
$ git commit -m "申请时的死锁问题的修正"
$ git commit -m "删去没用的代码"
```

在 issue345 上提交了 3 个修正后，分支的状态如图 3.3 所示。

图 3.3 新建的故障处理用的分支



●……… 合并到 master

修正告一段落后，让我们把修改的内容合并到 master。只需切换到 master 并执行以下命令即可。

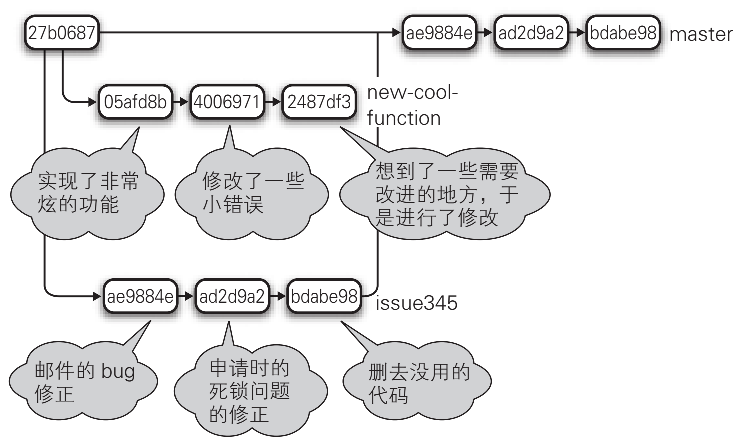
```
$ git checkout master
$ git merge issue345
Updating e380b5e..8f3b4e5
Fast forward
 src/main/scala/Application.scala |    1 -
 src/main/scala/MailSender.scala |    13 +
 2 files changed, 13 insertions(+), 1 deletions(-)
```

最终的状态如图 3.4 所示。

在完全不影响新功能开发的基础上，完成了对故障的处理[①]。

① 实际在案例分析中数据库的变更管理也是非常棘手的一个问题，需要予以解决。但这里暂且不进行说明，数据库变更管理的相关内容将在后面进行讲解。

图 3.4　合并到 master



●……… 向 master 进行 Push

接着，把本地机器上合并到 master 分支中的修正用`git push`命令 Push 到中央代码库。

```
$ git push origin master:master
```

这样 Push 就完成了。接着让 QA 部门进行验收测试，用`git checkout`命令就能回到原来的新功能开发。

```
$ git checkout new-cool-function
```

之后，在 new-cool-function 分支上进行新功能添加的开发，开发完成后合并到 master 即可[①]。

① 因为本次的例子非常简单，所以能够轻易地合并，但也有发生冲突无法合并的时候。在这种情况下，首先要合理地解决冲突，然后再重新合并。这和 Subversion 等其他的版本管理系统的思考方式是一样的。

●……… 分支使用方法总结

像这样通过合理使用分支功能，并行地进行多个版本的开发也变得容易了[①]。

Subversion 这样的中央集权型版本管理系统无法像 Git 这样简单地建立分支。Git 的话还可以在本地机器的封闭环境下随意地建立分支、进行合并，因此能够以简单且高速的方法实现并行开发。

3.5.2 标签的使用方法

●……… 什么是标签

在刚才的例子中，你是否觉得如果能把发布时的系统快照保存下来会很方便呢？标签（tag）就是用于实现这个功能的。实现的细节上可能有所差异，但几乎所有的版本管理系统都提供了标签功能。

●……… 新建标签

还是刚才的例子，首先可以为发布的时间点打上标签。下面就以“v0.1" 为标签名，执行`git tag`命令。标签名在内容上要简洁易懂，一般多以版本号作为标签名。

```
$ git tag -a v0.1 -m "最早的发布版本"
```

这样以后也能找出这个时间点的系统快照了。

第 2 章的案例分析中，发布之后已经过了一段时间，这样的情况下仍然可以追溯回去打上标签。首先在 master 分支上用`git log`命令寻找对应的提交。

```
$ git log --pretty=oneline
27b06870957e44d5b02606af668c9a120df4b7e0 最早的发布版本
ae9884e9f1e7551026c447556c63db58d49d6774 邮件的bug修正
ad2d9a217af750b2fb0a0636f0a26ef761ba2bfc 申请时的死锁问题的修正
bdabe98d3b9102ce33ed7b4c6033ebc9cbb44fe6 删去没用的代码
```

① 关于分支功能请参考 *Pro Git* 中文版“Git 分支”一章（http://git.oschina.net/progit/3-Git- 分支 .html）或者 nulab 提供的“猴子都能懂的 Git 入门”（http://backlogtool.com/git-guide/cn/ ）等，上述资料对 Git 的分支功能做了浅显易懂的总结。

找到对应的提交之后，用`git tag`命令为那个提交打上标签。这样就 OK 了。

```
$ git tag -a v0.1 27b06870957e44d5b02606af668c9a120df4b7e0
```

●……… 标签的确认

下面来确认下刚才建立的标签。

```
$ git tag
v0.1
```

确认了标签之后，再来确认下标签对应的内容是否正确。

```
$ git show v0.1
tag v0.1
Tagger: ikeike443 <ikeike443@gmail.com>
Date:   Thu May 16 15:32:59 2013 +0900

commit 27b06870957e44d5b02606af668c9a120df4b7e0
Author: ikeike443 <ikeike443@gmail.com>
Date:   Thu May 9 19:29:49 2013 +0900

    最早的发布版本
...
```

可以用`git show`命令来确认标签的内容。从结果来看内容似乎是正确的。在本地环境上打好标签后，用`git push`命令 Push 到中央代码库。

```
$ git push origin v0.1:v0.1
```

●……… 标签的取得

如果要在其他环境上取得刚才建立的标签，应该怎么做呢？

首先，在其他环境上用`git clone`命令进行克隆。`git clone`，顾名思义，就是克隆代码库的命令。代码库中包含的分支、标签都会被复制到本地环境中。

```
$ git clone git@github.com:CoolInc/someniceapp.git
```

克隆执行完后建立分支，并将标签 checkout 到该分支上[①]。

① 这里也可以把标签名和分支名都设置成“v0.1”，但这里没有这么做。相关原因请参考专栏。

```
$ git checkout -b 0.1 v0.1
```

这样就能在其他环境上取得标签了。如果不想新建分支的话，还可以用下面的方法[①]。

```
$ git checkout v0.1
```

专栏　避免使用相同的标签名和分支名

本文的例子中虽然避免使用相同的分支名和标签名，但使用相同的名称也是可以的。

```
$ git checkout -b v0.1 v0.1
```

新建名为“v0.1”的分支，在其基础上 checkout 名为“v0.1”的标签。但是因为同名的缘故，“v0.1" 指的是分支还是标签就不清楚了。

这里，因为执行的时候还没有名为“v0.1”的分支，所以能够顺利执行。如果已经存在名为“v0.1”分支，则是无法执行的。例如，如果之后再新建名为“test”的分支，并在其上 checkout“v0.1”标签，就会出现以下消息。

```
$ git checkout -b test v0.1
warning: refname 'v0.1' is ambiguous.
fatal: Ambiguous object name: 'v0.1'.
```

相反，在从 master 切换到“v0.1”分支时也会出现同样的问题。

```
$ git checkout v0.1
warning: refname 'v0.1' is ambiguous.
fatal: Ambiguous object name: 'v0.1'.
```

在这种情况下，需要像下面这样明确指出这个是标签还是分支。

```
//v0.1checkout 名为“v0.1”的标签
$ git checkout -b test refs/tags/v0.1

//v0.1checkout 名为“v0.1”的分支
$ git checkout refs/heads/v0.1
```

Git 中，标签和分支实际上分属于不同的命名空间，因此通常在使用时不需要特别指明是标签还是分支，可以省略。所以说，乍一

① 这样操作会变成 Detached HEAD 的状态。相关内容请参考专栏。

看标签和分支还是可以同名的。

但这样很容易造成混乱。实际使用中要避免标签和分支同名，这样混乱也会相应减少。万一出现同名的情况下，可以通过指定命名空间来明确是标签还是分支。请记住这一点。

●········ 标签使用方法总结

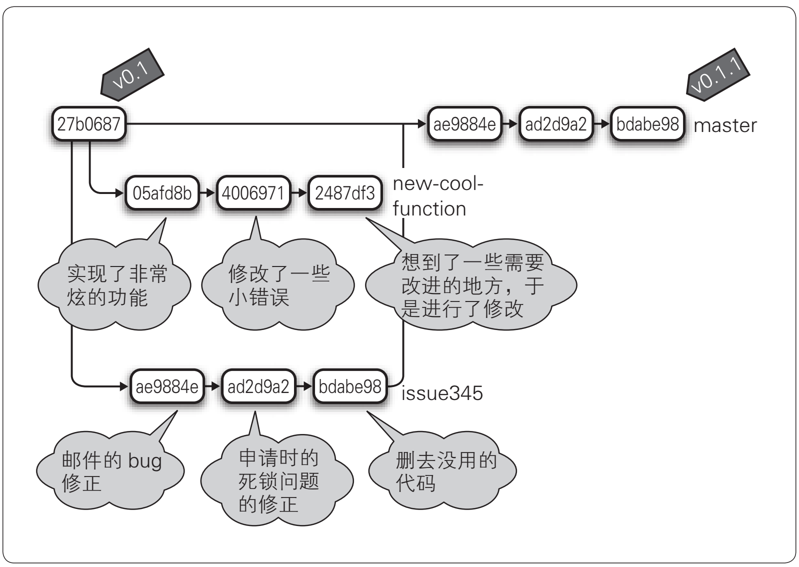
像这样，通过用标签管理系统某个时间点的快照，就能合理地进行版本管理，在发生故障后查找原因时，以及在紧急情况下的系统回滚时，也能发挥作用。

本次的例子中，在最初发布时打了一次标签。之后处理故障，将修改合并到了 master，因此待 QA 部门的验收测试结束，可以再次发布时，应该再次打上标签。

```
$ git tag -a v0.1.1 -m "进行了issue 345的故障处理"
```

最终，合理使用分支和标签功能之后，版本管理系统的状态如图 3.5 所示。

图 3.5　活用标签



至此，我们一起看了使用分支和标签功能的基本的团队开发的方法。下一节我们将研究使用分支功能的团队开发工作流程的各种形式。

专栏　什么是 Detached HEAD

执行本文中的命令，会显示下面这样的消息。

```
$ git checkout v0.1
Note: checking out 'v0.1'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

git checkout -b new_branch_name

HEAD is now at 8a4e89c... xxxx
```

这就是 Detached HEAD 状态。不必建立分支就能实验性地修改并提交，或者放弃修改。执行`git branch`命令，显示如下。

```
$ git branch
* (detached from v0.1)
  master
```

如果最终想保留在这个状态下进行的各类实验性质的提交的话，只需重新建立分支，并切换到该分支上，就能将提交保存下来。

```
$ git checkout -b new_branch_name
```

Detached HEAD 状态虽然是很方便的功能，但在部署过程中获取标签的时候，如果还保持 Detached HEAD 状态，管理上就很难理解。所以要像本文中的示例一样，一开始就新建分支，并将其 checkout。但具体做法还是要结合项目自身的情况进行探讨。

3.6 Git 的开发流程

到这里为止，我们一起了解了分布式版本管理系统所具有的灵活性，以及 Git 的分支、合并、标签功能的便捷和高速。

那么，在实际使用 Git 进行团队开发的过程中，应该采用怎样的开发流程呢？

以 Subversion 为代表的中央集权型版本管理系统中能执行的项目并不多，所以开发流程也自然而然地只有一种。另一方面，因为 Git 过于灵活，所以容易让人产生困惑，不知道应该采用怎样的开发流程。

老实说，Git 开发流程现在似乎还没有固定的模式。原因之一是 Git 不仅功能丰富而且非常灵活，即便不规定流程也还是能正常使用。本节将介绍比较常用的 Git 工作流的模式。

3.6.1 Git 工作流的模式

Git 的工作流大致有如下几种模式。

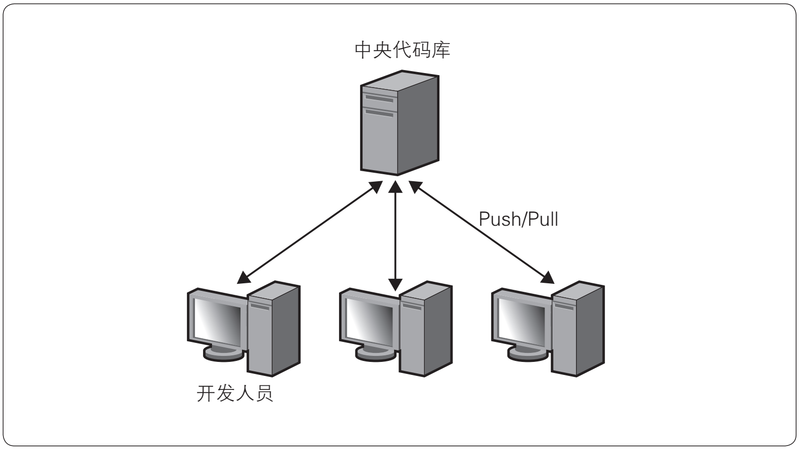
- **中央集权型工作流**
- **GitHub 型工作流**

中央集权型工作流

Pro Git[①] 中也举过这样的例子，即像以前的 Subversion 那样设置 1 个中央代码库，全体开发人员以此为唯一的 Push/Pull 对象，将环境克隆到本地机器上（图 3.6）。

① 该书第 2 版将由人民邮电出版社出版。——编者注

图 3.6 中央集权型工作流



该工作流的优点是操作和 Subversion 基本相同，思考方式也不需要太大的改变。另外，借助把 Subversion 的代码库视作 Git 来进行交互的工具 git-svn，实际的中央代码库可以仍然是 Subversion。当现有的代码库是 Subversion，并且移植到 Git 的成本很高时，这种情况下该工具是非常有效的。

GitHub 型工作流

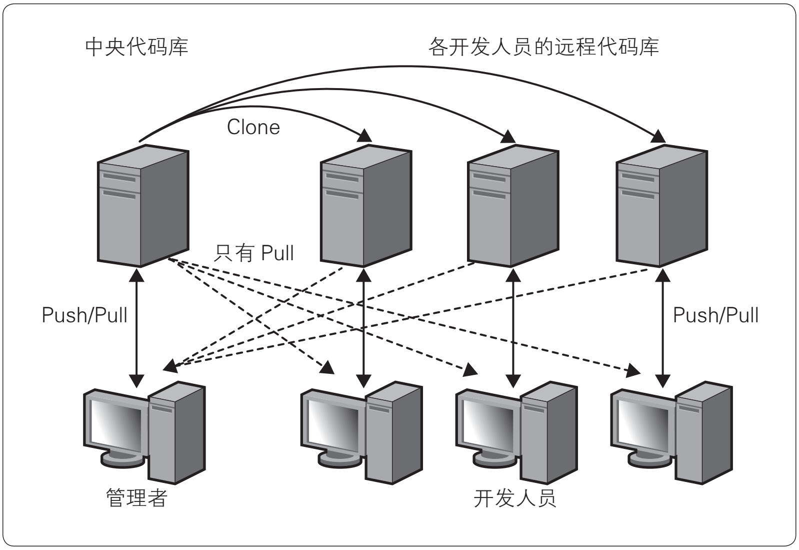
Pro Git 中称之为"整合经理型工作流"。因为该流程和 GitHub 的机制基本相同，所以本书称之为 GitHub 型工作流。

GitHub 型工作流同样也设置 1 个代码库为中央代码库，但开发人员会各自从中央代码库 Fork 出自己的远程代码库，并将其克隆到本地机器上。

GitHub 型工作流的方式是，在日常开发中将修改 Push 到 Fork 出的自己专用的远程代码库上，当修改相对稳定后，请中央代码库的管理者进行 Pull 操作（图 3.7）。

这样的流程基本就是 GitHub 的流程。其优点在于不需要等待中央代码库的维护或合并操作，各开发人员能够并行地进行作业。开发人员拥有自己独立的代码库，可以在代码库之间自由地相互 Push/Pull。这也可以说是只有分布式版本管理系统才具备的优点。

图 3.7 GitHub 型工作流



在 Git 的系统上应用该流程时，因为没有任何约束，所以运用中不注意的话容易造成混乱。如果想利用这种方式，推荐使用提供以这种方式调整后的工作流的 GitHub。

3.6.2 分支策略的模式

在已经设置中央代码库的情况下，关于发布时的发布分支应该如何处理、开发时是否要建立 topic 分支、修正 bug 的分支要怎样处理等分支运用的策略也有着各种各样的模式。

●········ git-flow

git-flow 是 2010 年 Vincent Driessen 在“A successful Git branching model”这篇博客[①]中提出的分支运用策略。同时也指为了支持该分支策略而编写的工具。

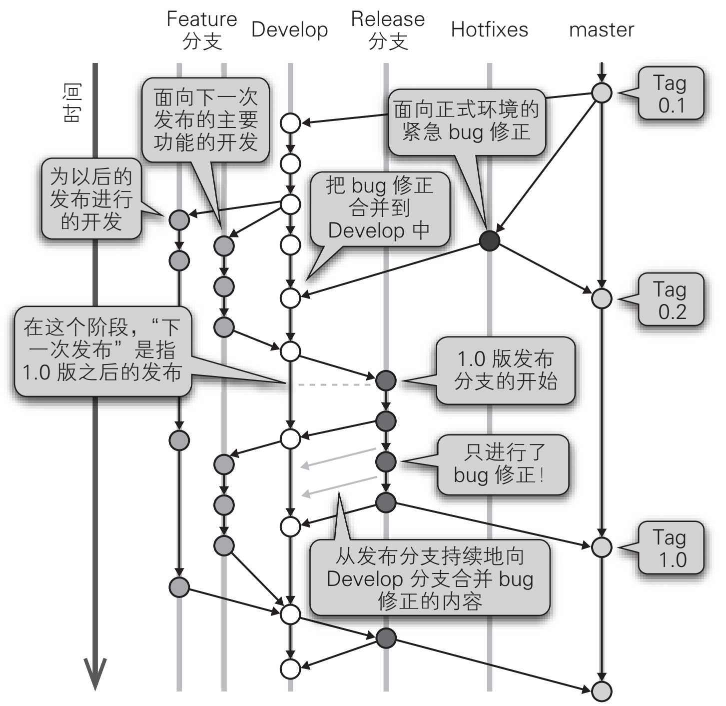
该分支策略在分布式版本管理系统的工作流中稍许引入了中央集权

① http://nvie.com/posts/a-successful-git-branching-model/

型版本管理系统的长处，可以说是结合了双方优点的团队开发流程。通过在团队内部统一管理建立分支的方法、合并的做法以及关闭分支的方法，来实现 git-flow 分支策略。

git-flow 提议以固定的模式在中央代码库中建立两个主分支，在各开发人员的代码库中建立 3 个辅助分支（图 3.8）。

图 3.8 git-flow



"A successful Git branching model" by Vincent Driessen（引自 http://nvie.com/posts/a-successful-git-branching-model/）

想详细了解该分支策略的读者请参考上述博客或相关资料。

图 3.8 中的各个分支的含义如下。

● 主分支

在中央代码库和从中央代码库克隆的各开发人员的代码库中一直存在的

主要的分支

- ○ Master

  为发布而建的分支。每次发布时都打上标签

- ○ Develop

  开发用的分支。发布之前的最新版本

- ● **辅助分支**

  原则上只存在于各开发人员的代码库中，是发挥相应的功能后就被删除的临时分支

- ○ feature

  从 develop 分离出来的被用于开发特定功能的分支。功能开发结束后被合并到 develop 中

- ○ release

  从 develop 分离出来的为发布做准备的分支。在发布的准备工作期间，为了避免多余的 feature 混杂到发布中而建立的分支。发布结束后被合并到 master 和 develop 中

- ○ hotfix

  主要是在发布后的产品发生故障时紧急建立的分支。直接从 master 分离，bug 修正后再合并到 master 并打上标签。为了避免将来遗漏这个 bug 的修正，还要合并到 develop。如果此时有正在发布作业中的 release 分支，还要向 release 进行合并

这样的分支策略条理清晰且容易理解，能够立刻投入到实际运用中。为了支持该分支策略，还提供有 Git 扩展脚本（git-flow 脚本）。通过使用脚本，运用会变得更加容易。

在有一定数量的人员的开发中，按照 git-flow 分支策略这样整理、运用分支，管理也会变得简单。

问题在于运用有些复杂，需要记忆的内容比较多。并且如果没有 git-flow 提供的 git-flow 脚本的话，运用会比较困难。特别是在使用 GUI 工具的情况下就无法沾到 git-flow 脚本的光了，这点需要注意[①]。

① Atlassian 提供的 SourceTree 工具好像支持 git-flow 分支策略。

●········ github-flow

github-flow 是由 *Pro Git* 的作者兼 GitHub 长官 Scott Chacon 在博客[①]中发表的 GitHub 的工作流。

Scott Chacon 也认为 git-flow 是非常出色的，但就实际运用比较复杂这方面，他列举了如下问题。

问题之一在于 Git 本身就绝非容易理解，此外还必须理解分支策略。

问题之二在于使用 GUI 工具的话就无法使用 git-flow 脚本，因此就不得不自己充分理解分支策略。何况难以理解分支策略的人往往也无法合理使用 git-flow 脚本，这样的人最终就无法合理运用 git-flow。

针对上述问题，GitHub 采用了下面这些更为简单的做法。这就是所谓的 github-flow。

- master 的内容都可以进行发布
- 添加内容时直接从 master 新建分支
- 建立的分支在本地环境上提交，并以同名的分支定期向远程代码库进行 Push
- 开发结束后向 master 发送 Pull Request
- Pull Request 通过 review 之后合并到 master，并从 master 向正式环境发布

还有一点在上述项目中没有提到，那就是 github-flow 是以使用 GitHub 进行开发为前提的[②]。

从博客的图片中可以看出，github-flow 没有 Fork 中央代码库，而是采用了代码库唯一的方式（中央集权型版本管理系统）。

该方式的特点在于简单、易于理解。开发人员只需要记住以下 3 点就行了。

- master 是用于发布的，不要直接在 master 上进行修改

① http://scottchacon.com/2011/08/31/github-flow.html

② 因为是 github-flow，所以这也是理所应当的。

- 开始作业时要先建立分支
- 作业结束后向 master 发送 Pull Request[①]

为了使 Pull Request 在通过 review 之后能立即进行合并并发布，在 Pull Request 发送之前需要在本地进行测试，或者保持使用 Jenkins 等 CI 工具的自动化测试一直运行，这是 github-flow 的前提。

当正式环境出现故障时，也只需要简单地从 master 建立分支并进行修正，发送 Pull Request 并合并，直接进行发布就行了。

像 GitHub 这样，通过保持 master 随时都能发布来简化实际的运用，这的确是一个不错的办法。GitHub 自身一天之内要进行几十次发布。为了实现这样频度的发布，就必须准备好 CI 以及 CD 的环境。而准备这样的环境是比较困难的。

●……… 笔者的例子（折衷方案）

git-flow 比较倾向于发布间隔较长的大规模项目，github-flow 则适用于需要时常发布的具有速度感的项目，例如 Web 服务等。下面举一个稍许极端的例子。

笔者所属的团队，因为 git-flow 比较复杂所以没有使用，实际运用的是定制过的 github-flow 形式。具体来说，就是像下面这样。

- 以使用 GitHub 为前提
- 采用 GitHub 形式的工作流，各自在 GitHub 上拥有 Fork 出来的远程代码库。
- 从中央代码库的 master 分支进行发布
- 各开发人员可以在自己的本地环境和远程代码库上随意地操作
- 开发结束后向中央代码库的 master 发送 Pull Request
- Pull Request 通过 review 后合并到 master
- 在发布准备阶段从中央代码库的 master 建立发布分支
- 在发布分支上进行回归测试、部署测试等发布的准备工作

① Pull Request 不仅在代码库之间，在同一代码库的分支之间也能发送。令人感到意外的是不知道这一点的人似乎还挺多的。

- **在进行发布的准备工作的过程中，开发人员仍然可以向 master 发送新功能开发的 Pull Request**
- **发布结束后合并到 master，在 master 上打上标签后删除发布分支**

大体的形式是只对中央代码库的 master 和发布分支进行严格管理，其他的都可以随意 Fork，自由使用。这种方式下的权限管理也变得相对简单，只需要对向 master 发送的 Pull Request 认真地进行代码 review 及测试，其余的事情就不用考虑了，这也是该方式的优点之一。

正式环境发生故障时的处理流程是：先将发布时打上的标签 checkout 到本地机器上并进行修正，修正后向中央代码库的 master 发送 Pull Request，合并 Pull Request 后建立发布分支，发布结束后将发布分支合并回 master 并删除。虽然比 github-flow 步骤稍微多了一些，但还并不是太复杂。

3.6.3　最合适的流程和分支策略因项目而异

本节我们了解了使用 Git 的开发流程以及分支策略。Git 系统中没有中央代码库的概念，因此能组建各种形态的工作流程。这样的灵活性正是 Git 的魅力所在，但过度的自由事实上也会造成很多麻烦。

接着我们简单地看了具有代表性的流程和分支策略。还介绍了笔者实际运用的例子。具体运用方面在很大程度上会受到团队的商业模式的影响。Web 服务的话，只需要时常管理好最新版本和已经发布的版本这两个就行了。而如果是套装软件的话，就必须同时维护多个版本，所以还是有所差异的。

另外，在发布分支和开发分支的操作方面，如果该产品是不经过长时间的回归测试就不能发布的产品的话，还是将发布分支和开发分支区分开来进行管理比较好。另一方面，如果是应该尽早发布的服务的话，则可以说像 github-flow 那样，完全不区分比较好。

Git 的开发流程能够很灵活地定制，所以请大家一定要结合项目的具体情况，试着思考一下最合适的流程。

3.7 数据库模式和数据的管理

3.7.1 需要对数据库模式进行管理的原因

团队开发中，如何才能有效地管理数据库模式[①]是一个比较棘手的问题[②]。在多人修改数据库的情况下，容易因漏执行 SQL 或执行顺序出错等原因造成数据库的数据一致性出现问题。

怎样才能解决这个问题呢？由于该问题的主要原因在于各个开发人员随意制作 SQL 修改数据库模式，因此应禁止开发人员修改数据库模式，并设置数据库管理员一职，由此人对所有的修改进行管理，这样是不是就能够解决问题了呢？

过去采用这种方式的开发现场比较常见，但从以下这些原因可以看出这种方式是存在问题的。

●……… 由数据库管理员负责对修改进行管理的情况

首先，在由数据库管理员负责对所有的修改进行管理的情况下，如果数据库管理员成为瓶颈的话，整体的开发速度就会受到影响[③]。

●……… 修改共享数据库的模式的情况

其次，如果在团队中共享数据库，那么模式的变更就会波及到整个团队，影响开发进度。模式发生变更时，首先程序的代码也无疑会发生变化，若修改共享数据库的模式，各个开发成员的代码和数据库模式就难免会发生背离。因此可以说对数据库模式也应该进行版本管理。

---

① 是对数据库构造的定义。

② 以第 2 章的问题 4 为例。

③ 根据项目的规模，设置数据库管理员也是必要的，因为有数据库的调整或备份等本来应该由数据库管理员负责的业务。

3.7.2 应该如何管理数据库模式

对数据库模式进行版本管理，应该管理什么？又怎么管理呢？让我们具体地来看一下。这里假设数据库为 MySQL 或 PostgreSQL 等，也就是说使用了 RDBMS，以此为前提来继续下面的话题。但是这里的数据库并不局限于 RDBMS，文本文件、XML 文件、对象数据库以及最近使用频率逐渐增加的 MongoDB 等 NoSQL 数据库，它们的思考方法也是完全相同的。

●……… 版本管理的必要条件

对数据库模式进行版本管理的必要条件中，比较重要的是以下 3 个。

- **无论什么环境都能用相同的步骤来构建数据库**
- **能够反复执行多次**
- **文本文件**

上面这些也是和 CI 相关联的思考方法。CI 相关的内容将在第 5 章进行说明。对于数据库模式，和代码一样进行版本管理，无论任何环境都能反复构建，这一点是非常重要的。另外，为了用版本管理系统方便地进行合并，以文本文件的形式管理模式也是很重要的。

例如有的开发现场使用商用的 GUI 工具来建立数据库模式，这样的工具有时反而会影响团队开发的效率。因此一定要以程序能够反复执行的文本文件形式来管理数据库模式。

●……… 什么是数据库迁移

数据库模式的 CI 称为 CDBI（Continuous DataBase Integration）。《持续集成：软件质量改进和风险降低之道》[①]中也以专门的章节对其进行了说明。但是最近比起 CDBI，使用从 Ruby on Rails 的工具名（Migration）衍生而来的“数据迁移”这个叫法的人似乎更多一些。本

① （美）Paul M. Duvall 、Steve Matyas、Andrew Glover 著，王海鹏译，电子工业出版社，2012 年。

书也以数据迁移这个用语来继续下面的解说。

如今发布了很多数据迁移用的工具和框架，借助这些工具，能比较简单地实现数据迁移。

●……… 数据库迁移的功能

如前所述，作为数据库迁移工具，Ruby on Rails 的 Migration 是比较有名的。顺便说一下，笔者所在的公司是自己开发原生的工具进行数据迁移的。的确，Ruby on Rails 从诞生以来也只经过了 10 年左右的时间，还算是比较新的架构。业务持续了 10 年以上的公司中，独自开发工具进行数据迁移的情况还是比较多的。

不同的工具在实现的细节方面有细小的差异，但基本的构思几乎都是相同的。笔者所在的公司开发的工具和以 Ruby on Rails 的 Migration 为代表的工具大体上都是以下面这样的构思来制作的。

- 管理 SQL 执行的顺序和需要执行哪些 SQL
- 管理模式定义编辑的冲突
- 提供回滚的机制
- 支持数据的加载

SQL 的模式定义（DDL，Data Definition Language）和数据加载（DML，Data Manipulation Language）是有执行顺序的。如同我们在第 2 章的问题 4 中所见到的那样，执行顺序发生变化后意义也会随之改变，所以需要对执行顺序以及在各个环境中需要执行哪些 SQL 进行管理。因此大部分工具都在数据库中建立管理表，或者用专门的 XML 文件来管理执行顺序和需要执行哪些 SQL 等。

为了避免冲突，通常会为 SQL 文件确立某种命名规则。具体来说，有的以日期作为文件名，将相同日期的 SQL 全部放入同一文件；有的以连续的数字作为文件名，文件名相同的话则进行合并等。

受 Ruby on Rails 的 Migration 的影响，如今的工具基本都提供了回滚的机制。制作文件时会在同一文件中记载回滚用的 SQL。CREATE 语句对应 DROP，INSERT 语句对应 DELETE，这样执行用的 SQL 和回滚

用的 SQL 就可以成对地进行管理。

数据加载方面和模式定义相同，比较多的工具采用在文件中记载 INSERT 或 UPDATE 的方法进行管理。并且这里的数据主要是系统初始化设置所必需的数据。例如税率的数据或超级用户的数据等。可以理解为和用户的使用状况无关，系统启动时所必需的数据。

请注意这里加载的数据不包括单体测试和结合测试所需要的数据。测试数据不属于初始化数据，而属于用户数据。这部分数据应该通过 Fixture[①]等机制在测试程序中加载。测试部分的数据管理将在第 5 章进行说明。

◆◆◆

接下来将对主流的数据库迁移工具进行说明。在因为某些原因无法使用工具的情况下，可以考虑自己制作符合要求的原生工具。通过对数据库模式实行版本管理，项目的品质和速度将会有突破性的飞跃。

3.7.3　数据库迁移工具

现在已经出现了各种各样的数据库迁移工具，并且几乎所有最近的 Web 应用程序框架中都自带数据库迁移工具。Web 应用程序框架自带的工具主要有以下这些。

●········ Migration（Ruby on Rails）

现在 Web 应用程序框架的先驱 Ruby on Rails 自带的工具。其他的工具几乎都受它的影响。为了管理 SQL，会在数据库中建立名为 system_setting 的表。

●········ south（Django）

比较有名的 Python 框架 Django 中自带的名为 south 的工具。

① 将测试所需要的数据预先加载到数据库中的机制。为了能够反复测试，通常 Fixture 会在执行测试前加载数据，在测试结束后删除数据。像这样预先提供数据加载和删除这两个功能的情况是比较多见的。

●········ Migrations Plugin（CakePHP）

PHP 的框架 CakePHP 中为了进行数据迁移而提供的插件。

●········ Evolution（Play Framework）

Java 和 Scala 的框架 Play Framework 中作为标准自带的名为 Evolution 的工具。

◆◆◆

也有不局限于特定的 Web 应用程序框架的通用迁移工具。这类工具的数量比较多，我们这里只介绍一小部分能够用 Java 调用的工具。

- Flyway①
- Liquibase②
- dbdeploy③

每个工具的功能都差不多，所以选择与项目相适应的，可以使用的工具就行了。

如果要在已有的应用程序中导入迁移工具，那么使用不依赖于框架的工具会比较好。还有一些 Java 的项目，只是为了使用 Migration 而特地导入 Ruby on Rails，这么做的开发现场也是有的。

3.7.4　具体用法（Evolution）

关于数据迁移工具的用法，这里以笔者作为提交者（committer）的 Play Framework 为例来进行说明。Evolution 和老大哥 Migration 比起来功能上要简单很多，但必要的功能都有了。

●········ 规定

Evolution 和其他类似的工具一样，对于文件名、记载的内容、存放

① http://flywaydb.org/
② http://www.liquibase.org/
③ http://dbdeploy.com/

的目录等都有规定。

- SQL 文件以 1.sql、2.sql 这样连续的数字命名
- 一定要放在程序的 conf/evolutions/{ 数据库名 } 的目录下
- SQL 文件中必须要有 Ups 和 Downs[①]这两部分的定义

例如像下面这样定义 1.sql 文件[②]。

```
# User表

# --- !Ups

CREATE TABLE User (
    id bigint(20) NOT NULL AUTO_INCREMENT,
    email varchar(255) NOT NULL,
    password varchar(255) NOT NULL,
    name varchar(255) NOT NULL,
    creat_at date NULL,
    update_at date NULL,
    PRIMARY KEY (id)
);

# --- !Downs

DROP TABLE User;
```

●……… SQL 文件的执行

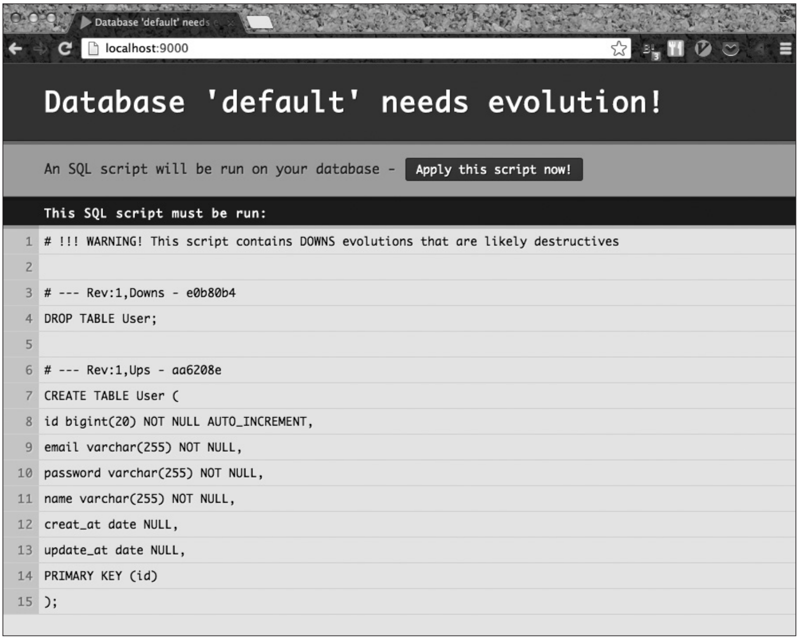
Play Framework 在存在 SQL 文件的状态下启动程序就会自动执行 Evolution 的处理[③]。处理会对比本地机器上的数据库和 Evolution 的 SQL 文件，如果发现有没有被执行的 SQL，就会在浏览器中显示如图 3.9 这样的消息，提示执行 SQL。

① Ups 部分记载需要执行的 SQL 的定义，与之相对，Downs 部分记载的是将 Ups 部分的内容回滚时需要执行的 SQL。如果 Ups 部分是 Create 的话，Downs 部分就是 Drop，大致就是这个样子。详细的处理稍后进行讲解，大概机制是当版本管理系统中的 SQL 和实际的数据库状态之间的一致性出现问题时，就运行 Downs 部分的内容进行回滚。

② Ruby on Rails 的情况下不是 SQL 文件，而是用 Ruby 写的单独的 DSL 文件。

③ 当然也可以设置为不自动执行。

图 3.9　有 SQL 没有被执行时的画面（Play Framework）



按下“Apply this script now!”就会执行 SQL 文件。关于这方面，Ruby on Rails 的 Migration 采用的是命令行操作模式，Play Framework 采用的是对话模式[①]。交互模式虽然不同，但想要处理的内容是相同的。

●……… 开发者之间数据库模式的同步

假设开发人员 A 为添加表而新建了 2.sql 文件。

```
# Company 添加

# --- !Ups
CREATE TABLE Company (
    id bigint(20) NOT NULL AUTO_INCREMENT,
    name varchar(255) NOT NULL,
    location varchar(255) NOT NULL,
    description text NOT NULL,
    url varchar(255) NOT NULL,
```

① 最新的 Ver2 只能使用对话模式进行处理。Ver1 也可以用命令行模式进行操作。希望 Ver2 的 Evolution 也能早日支持命令行模式。

```
    creat_at date NULL,
    update_at date NULL,
    PRIMARY KEY (id)
);

# --- !Downs
DROP TABLE Company;
```

在 A 的数据库中执行该 SQL。

这时开发人员 B 正好想给 1.sql 中建立的 User 表增加一列。因为不知道 A 已经建立了 2.sql，所以 B 在本地机器上又建立了 2.sql。

```
# User 修改

# --- !Ups
ALTER TABLE User ADD age INT;

# --- !Downs
ALTER TABLE User DROP age;
```

开发结束后，B 向中央代码库 Push 新建的文件。之后 A 也进行 Push，但因为 B 提交的 2.sql 已经存在，所以结果产生冲突了。

```
<<<<<<< HEAD
# Company 添加

# --- !Ups
CREATE TABLE Company (
    id bigint(20) NOT NULL AUTO_INCREMENT,
    name varchar(255) NOT NULL,
    location varchar(255) NOT NULL,
    description text NOT NULL,
    url varchar(255) NOT NULL,
    creat_at date NULL,
    update_at date NULL,
    PRIMARY KEY (id)
);

# --- !Downs
DROP TABLE Company;
=======
# Update User

# --- !Ups
ALTER TABLE User ADD age INT;

# --- !Downs
```

```
ALTER TABLE User DROP age;
>>>>>>> devB
```

合并后的代码如下所示。

```
# Company 添加

# --- !Ups
ALTER TABLE User ADD age INT;

CREATE TABLE Company (
    id bigint(20) NOT NULL AUTO_INCREMENT,
    name varchar(255) NOT NULL,
    location varchar(255) NOT NULL,
    description text NOT NULL,
    url varchar(255) NOT NULL,
    creat_at date NULL,
    update_at date NULL,
    PRIMARY KEY (id)
);

# --- !Downs
ALTER TABLE User DROP age;

DROP TABLE Company;
```

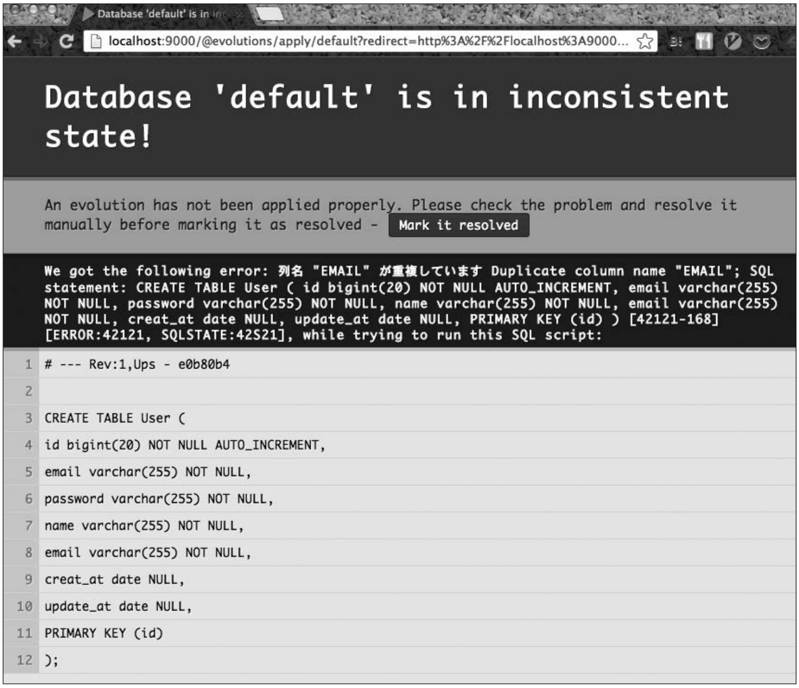
合并后新的 2.sql 和执行了旧的 2.sql 的开发人员 A 的数据库之间已经有了差异。Evolution 会检查出该差异，回滚掉旧的数据库模式，并重新建立新的模式。回滚处理时会用到 Downs 部分，回滚到执行 2.sql 之前的状态后，再重新执行 Ups 部分。在此之后，如果 B 进行 Pull，B 的数据库也会被回滚，然后被重新建立。

这样，多人各自持有的数据库模式就都得到了同步。

一致性问题的管理

由于某些原因，执行 SQL 也会有失败的时候。例如合并 SQL 时出现问题，生成了自相矛盾的 SQL，并且没有注意就执行了该 SQL。这时 Evolution 会检测出错误，作为模式不一致的状态进行管理（图 3.10）。

图 3.10　一致性问题的管理（Play Framework）



显示上述画面时，可以通过手动执行 SQL 进行修正后再按下 `Mark it resolved`按键，或者修改原有的 SQL 文件后再运行 Evolution 的方法来解决。

这样，通过对 SQL 的错误进行管理，使得管理正确的数据库模式成为了可能。

3.7.5　数据库迁移中的注意点

对数据库迁移文件（Evolution 的话是 SQL 文件，Migration 的话是 Ruby 文件）进行版本管理时，判断合并的结果是否正确是一个比较伤脑筋的问题。

RDBMS 的数据库构建可以说只要不出现错误就应该没什么问题，但无法保证合并后的 SQL 一定正确，因此需要对数据库模式的正确性

进行测试。

这个和代码的合并道理相同。代码也是合并后的文件只要能通过编译就基本可以说没问题了，但从程序的角度来看，正确与否还要经过测试才知道。

只要使用数据库迁移，数据库的变更管理就万无一失，这样的说法过于草率，测试还是必须的，这一点请注意。应该编写测试程序，做到随时都能够自动进行测试。因此应该将数据库迁移作为 CI 的一个环节，纳入到 CI 之中。CI 的相关内容将在第 5 章进行说明。

3.8 配置文件的管理

如同我们在第 2 章的问题 10 中见到的那样，由于环境不同，对依赖环境的各种配置文件也要进行版本管理。首先，程序方面的配置应该把每个环境的配置文件分开进行管理。例如数据库或 memcached 等中间件的连接地址配置[①]或程序本身固有的配置等。现代的 Web 程序框架一般都会提供这样的机制，请直接使用。即便出错了，也不要采用手动替换 staging 环境或正式环境中的配置文件这样的方式。

将所有资源都置于版本管理系统之下，只使用版本管理系统中的文件，任何人都可以随时自动化地执行程序的构建到启动的所有过程，只有这样才能维持优质、快速的开发。

接着是 Apache、MongoDB、memcached、PostgreSQL 等中间件自身的配置文件的管理。这些中间件基本上都没有根据不同的环境分别进行配置的功能。因此大多数的公司都独立开发构筑服务器用的安装脚本，或者在目录结构和符号链接方面下功夫，以尽可能地把中间件的安装到配置的过程简单化。

通常部署程序的服务器由多台组成，如果不能有效地管理配置文件，要进行快速的发布是不可能的。现在用于解决上述问题的工具有 Chef、Puppet、Capistrano、Fabric、ServerSpec 等。使用这些工具的管理方法将在第 6 章进行说明。

① 以数据库为代表的中间件的连接信息中一般会包括密码。将密码这样的信息直接提交到版本管理系统的代码库可能会产生安全方面的问题，这点需要注意。通常只将开发环境的密码提交到版本管理系统，不提交 staging 环境或正式环境的密码，而使用部署脚本在每次发布时进行密码设置。

3.9 依赖关系的管理

开发具有一定规模的程序时，难以避免地会使用到一些库。程序所依赖的库的依赖关系的定义也应该作为版本管理的对象。

3.9.1 依赖关系管理系统

现在各开发语言一般都会提供以下 3 点的组合。

- **管理通用库的仓库**
- **定义对库的依赖关系的文件**
- **使用上述文件实际管理依赖关系的脚本**

下面列举各开发语言提供的一些主要工具。

JVM 语言

为了管理 Java 或 Scala 等在 JVM 上运行的语言的依赖关系，有着各种各样的工具。

- Apache Ant[①]（+Apache ivy[②]）
- Maven[③]
- sbt[④]
- Gradle[⑤]

① http://ant.apache.org/
② https://ant.apache.org/ivy/
③ http://maven.apache.org/
④ http://www.scala-sbt.org/
⑤ http://www.gradle.org/

这些工具都参照下面 Maven 的仓库作为共通的仓库。

- Maven 中央仓库[①]
- Sonatype（中央仓库镜像）[②]

如果制作了可以作为 OSS 公开的通用库的话，只要上传到上述仓库，就能在全世界范围内共享。

还可以在公司内部构建 Maven 的仓库。只想在公司内部共享的库等，可以上传到公司内部的 Maven 仓库中进行管理。

●……… 脚本语言

每个脚本语言都有自己各式各样的依赖关系的管理工具。

- CPAN[③]（Perl）
- PyPI[④]（Python）
- RubyGems[⑤]（Ruby）
- npm[⑥]（Node.js）

和 JVM 语言不同，脚本语言的仓库和工具是相同的。这是因为脚本语言较早地采用了包（package）管理的思考方式，并一直在进行整理。另一方面，JVM 语言因为长时间没有采用这种思考方式，所以各类工具显得有点混乱。

●……… 管理依赖关系的优点

例如 Maven 中的依赖关系可以像下面这样定义。

```
<dependencies>
  <dependency>
```

① http://search.maven.org
② https://oss.sonatype.org
③ http://www.cpan.org/
④ https://pypi.python.org/pypi
⑤ http://rubygems.org/
⑥ https://npmjs.org/

```
    <groupId>org.jyaml</groupId>
    <artifactId>jyaml</artifactId>
    <version>1.3</version>
  </dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
  </dependency>
</dependencies>
```

这样定义后，Maven 会自动地从仓库中找出 jyaml 和 junit，并将其加到 ClassPath 中。如果只是处理这些的话，你可能会觉得这和在网上查找库并手动下载，然后再添加到 ClassPath 没什么区别。事实上并非如此。

不仅是 Maven，上述列举的各个工具都还能够管理传递依赖。具体来说，如果 jyaml 和 junit 自身有依赖的库的话，就会对其进行查找，找到的库如果还依赖其他的库，则继续查找下去。也就是说，这些工具具有自动管理所有传递的依赖关系的机制。

这样，依赖关系也能作为可以重复执行的内容被纳入到版本管理的对象之中了。如果手动操作的话，不仅容易遗漏库，再加上环境差异等原因，也非常麻烦。因此在开发现场依赖管理工具是必不可少的。

上面以 Maven 为例进行了说明。其实 CPAN 也好，RubyGems 也好，基本的思考方式都是一样的。在接下来的开发中，让我们一起使用依赖管理系统进行有效地管理吧①。

① 各个工具的详细用法在 Web 上有很多相关的信息可供参考。

3.10 本章总结

至此我们一起确认了什么是版本管理系统，一边回顾版本管理系统的历史，一边介绍了如今的开发现场所面临的问题。还了解了分布式版本管理系统的使用方法、版本管理系统应该管理的内容以及有效的管理方法。

这样就能够记录下什么时候、谁、进行了怎样的修改，任何时候都可以进行追查。并且还能够做到根据需要回滚到过去某一时刻、多版本并行开发、有效地进行数据库的模式变更管理。

但是只有版本管理系统的话还是不知道为什么要进行这样的修改，解决这个问题就需要用到缺陷管理系统。

在接下来的第 4 章，我们将一起看一下缺陷管理系统的有效用法，特别是和版本管理系统联动的方法。

第4章

缺陷管理

4.1 缺陷管理系统

4.1.1　项目进展不顺利的原因

如果参加项目的多名成员同时还兼任其他工作，那么在推进项目的过程中重要的就是任务整理、进度管理和信息共享。

项目成功 / 失败的形式各种各样，并且不同的项目和团队对于成功 / 失败的定义也各不相同。并不是完成了所有的任务就是成功。例如，即使所有的任务都按时完成了，但如果最终没有到达当初制定的目标，不也可以称之为失败吗？

话虽如此，项目能够进展到可以评判成功 / 失败的阶段可以说还算不错的了。在这之前，有的项目就已经出现“进展不顺利、结束不了”的情况了。笔者有过很多这样的经历，相信读者也有类似的记忆吧。

项目“进展不顺利、结束不了”的原因各种各样，大致有下面这些。

- **目标错误**
- **估计错误，时间过紧，人员不足**
- **没有定义项目的结束**
- **成员的积极性不足，进度停滞不前**

从笔者的经验来看，“进展不顺利、结束不了”的项目有着以下这些共同的特征。

- **项目无法可视化**
- **没有进行任务整理、进度管理和信息共享等**

换言之，这样的项目多数都没有做到“必须要做什么”这样的任务定义、由“谁”在“什么时候完成”这样的责任人和期限的管理、“以什么作为结束”这样的任务完成的定义，以及“现在正在作业中还是已

经完成了"这样的进度管理。

但是不进行任务管理的项目真的存在吗？毕竟笔者还没有听说过这样的事情，无论哪个现场都应该以某种形式对任务进行管理。那为什么还会有进展不顺利的项目呢？这可能是因为管理方法的问题。

4.1.2 用纸、邮件、Excel 进行任务管理时的问题

最近，随着缺陷管理系统的导入，像第 2 章问题 1 那样用邮件管理全部内容的开发现场已经越来越少了。

即便如此，不使用缺陷管理系统也能够实现任务管理、进度管理和信息共享。年轻的读者之中可能有人不知道，在过去的开发现场，进度管理是用名为"bug 票""问题管理表"这样纸质的管理票和账本来进行的。还有的开发现场用 Excel 制作问题管理表来进行管理。如果是小规模的开发现场，用邮件进行任务管理也不是不可能的。

但是随着项目涉及的人数和任务数量的增加，这样的管理手法就会出现问题。用纸来管理既影响开发速度又不适合在多人之间共享信息。

可能有人认为用 Excel 进行管理应该还是可行的[①]，但因为 Excel 自身并不是专门的问题管理工具，所以在稳定性等方面存在问题。用 Excel 制作问题管理表，并将其存放在共享目录中进行管理，多人同时编辑后文件就损坏了，这样的经历相信大家都有过吧[②]。

还有一些开发现场使用的是 Microsoft Project[③]。该产品适用于项目经理从高层次的视角向利益相关者汇报项目的情况，并不太适合现场的作业管理。和 Excel 相同，在多人同时编辑时稳定性方面存在问题。除此之外，纸、邮件、Excel 等还有其他欠缺的地方，那就是信息的统一管理和检索。

对于后来加入项目或团队的成员来说，过去的事情以及需求等信息统一记录在一处，并且能够进行检索是非常重要的。如果能够做到这一点的话，之后加入的成员就能够进行自学，加快追赶的速度。最终项目

① 实际上设法用 Excel 进行管理的开发现场不在少数。

② 无法用邮件进行管理的原因在第 2 章说明过了，这里就不再提了。

③ 微软开发的项目管理软件程序。

的成功率也会相应地上升。

多数的缺陷管理系统都附带有 Wiki 的机制，该机制能够在信息共享方面起到作用。例如使用问题票对课题进行调查和交流，最后把调查获得的知识以及确定下来的需求等总结记录在 Wiki 中。这样做能够提高信息的精确度，提高项目和团队的工作效率。

4.1.3　导入缺陷管理系统的优点

可能和之前的说明有所重复，这里让我们从软件开发项目这一大背景出发，再来确认下导入缺陷管理系统的优点。

●……… 具有任务管理所需的基本功能

缺陷管理系统的显著优点是具备下列这些功能。

- **“必须要做什么”这样的任务定义**
- **“谁来做”这样的职责分配**
- **“什么时候完成”这样的期限管理**
- **“现在正处于作业中还是已经完成了”这样的状态管理**

另外，作为项目中专门的任务管理程序，多数缺陷管理系统都是以多人使用为前提进行设计的，因此高稳定性及高可靠性也是其主要的优点之一。

●……… 直观性、检索性较强

在管理复杂项目的过程中，将项目中的问题、任务列表以各种形式显示出来是非常重要的。另外，能够检索找到过去的各类处理记录也是非常重要的。

●……… 能够对信息进行统一管理及共享

过去的处理记录、从中获得的知识，以及项目的需求等对于项目来说都是非常重要的信息，将这些信息统一管理并共享是非常重要的。这

些功能加上之前提到的高检索性，使得缺陷管理系统超越了单纯的任务管理系统的范畴，还具备了知识数据库的特性。

●……… 能够生成各类报表

统一管理数据所带来的显著优点之一就是易于生成各类报表。例如每周生成的在进度会议上要使用的工作完成情况报告、能够一目了然地掌握团队状态的燃尽图，以及向顾客说明时使用的甘特图等，缺陷管理系统能够方便地生成各式各样的报表。

能够以怎样的程度生成怎样的报表，这方面不同的缺陷管理系统会有所差异。有些是作为系统的基本功能提供的，有些是以插件的形式提供的。大多数的系统都提供了数据下载功能，如果没有符合需求的报表，可以下载数据后使用电子表格等软件自己生成报表。如果系统提供了 API 或插件系统，还能够用其来自由地开发报表功能，这部分内容稍后会详细介绍。

随着团队开发的推进，需要报表的情况会越来越多。强大的报表功能就意味着制作报表所用的时间能大大减少，这也是导入缺陷管理系统的显著优点之一。

●……… 能够和其他系统进行关联，具有可扩展性

特别想强调的就是这一点。随着缺陷管理系统和版本管理系统以及持续集成（Continuous Integration，CI）系统之间的关联的逐渐深入，从问题的发生到代码的修改内容、测试结果、发布情况等所有的内容都可以相关联地进行管理。

通过使用该功能，就可以从版本管理系统的提交记录追溯到原本的问题票，确认原本的修改理由，或者倒过来检索过去的问题找到对应的问题票，再从问题票找出相关联的提交和测试结果，以了解该问题是如何解决的。这样就能够追溯到项目内的活动的各个方面，使得可追溯性有飞跃性的提高。具体方法将稍后叙述。

具备和外部系统关联的可扩展性也是导入缺陷管理系统的显著优点。大部分的缺陷管理系统都具备插件功能或者能够从外部操控问题票

的 API 接口，因此能够满足项目的各种需求。

4.1.4 什么是缺陷驱动开发

使用缺陷管理系统，以问题票（ticket）为中心构建开发流程的方法论称为缺陷驱动开发（TiDD）。这个名字是仿照测试驱动开发 TDD（Test Driven Development）这一敏捷开发的方法而取的。

实践缺陷驱动开发的规则原则上只有一条，那就是“禁止没有问题票的提交”。提交记录和问题票必须一对一地进行管理，以此来明确代码修改的理由，这便是缺陷驱动开发的目的所在。

●……… 缺陷驱动开发的具体步骤

缺陷驱动开发每次都是以在新功能开发和 bug 修正时新建记有内容和目的的问题票开始的。

其次是根据问题票的内容对代码进行修正并提交。这时在提交记录中记下问题票号是非常重要的，同时还要在问题票中记录下提交记录号。这样问题票和代码修正提交之间的相互关系就建立起来了，明确了代码的修改理由，日后进行问题调查也变得容易了。

缺陷驱动开发是方法论，对工具的使用并不是必需的。如果只是关联提交记录和问题票的话，手动作业也是可行的。实际上手工关联提交记录和问题票的开发现场并不在少数。

虽说如此，但如果能够合理使用缺陷管理系统的话，关联作业也可以实现自动化。这样就能避免操作遗漏，准确地关联提交记录和问题票，进而促进项目的可视化。

从下节开始将主要介绍缺陷管理系统，说明其同版本管理系统关联的方法。另外，应该和缺陷管理系统关联的并不仅限于版本管理系统。在和提交相关联并实行 CI 的情况下，CI 的结果也要记录到问题票中，以方便日后调查。因此还会讲解和 CI 系统关联的相关内容。

专栏 彻底贯彻缺陷驱动开发的情况

上文中提到了在缺陷驱动开发的实践中工具的使用并不是必需的，手工关联问题票和提交也是可行的。话虽如此，手动作业的话容易发生问题票和提交之间的关联遗漏。

要想彻底贯彻缺陷驱动开发，可以使用版本管理系统提供的客户端挂钩（client side hook）和服务器端挂钩（server side hook）。在上述挂钩中能够检查提交信息中是否包含问题票号，如果不包含的话则拒绝接受该提交（或 Push）。

这样就可以系统性地保证问题票和提交之间的关联，原因不明的修改将无法提交到代码库，项目也就更安全了。并且之后回顾修改时也一定能找到作为修改原因的问题票，这点在管理上非常有优势。

但是在管理方面的优势有所增加的同时，反过来开发的速度就可能受到影响。这方面需要权衡考虑，要根据项目的目的和状况进行平衡。

4.2 主要的缺陷管理系统

缺陷管理系统[①]包括 OSS（Open Source Software）以及商用软件在内，有很多可供选择的产品。这里将介绍一些具有代表性的产品[②]。

4.2.1　OSS 产品

Trac

Trac[③]是基于 Python 的缺陷管理系统（图 4.1）。它导入简单，印象中过去几乎所有的开源项目都使用 Trac，之后被后来的 Redmine 以及 SaaS（Software as a Service）形式的 GitHub 和 PivotalTracker 等逐渐取代，最近已经很少看到了。但如果要说上手简单的话，Trac 至今依然是第一选择。

图 4.1　Trac



① 也可以称为 bug 管理系统或问题管理系统。

② 产品的具体功能请参考相关资料。特别是商用产品，因为其价格、服务形式以及提供的功能等可能发生变化，所以请一定要参考供应商提供的信息。

③ http://trac.edgewall.org/

Trac 能够和 bug 管理及代码管理相关联，具备基于 Wiki 的信息共享等项目所需要的功能，还可以通过插件进行各类扩展。

Trac 已经有了日语版的安装包。日本的开源社区也相当活跃，还出版了相关的书籍[①]。特别是由志愿者提供的 Trac Lightning[②]在日语版 Trac 的基础上还加入了 Subversion 和 Jenkins，将它们合并成一个安装文件，初次使用时非常方便。Trac Lightning 是 Windows 平台的安装包，Linux 平台的话可以使用由相同人员开发的 Kanon[③]。

Trac 原本是作为程序员的 bug 管理系统而开发的，因此用户界面比较简陋。如果团队中有程序员以外的成员，Trac 那过于简陋的接口就可能成为导入 Trac 的障碍。

●········ Redmine

Redmine[④]是比 Trac 稍晚的基于 Ruby on Rails 框架的缺陷管理系统（图 4.2）。因此，使用 Ruby on Rails 的项目导入 Redmine 的门槛会比较低。

Redmine 同样具备了 bug 管理、代码管理、Wiki 等必要的功能，支持插件系统，扩展也比较容易。UI（User Interface）方面要比 Trac 更为易用、漂亮，非程序员使用起来也基本上没有障碍。

Redmine 的日语开源社区规模大、非常活跃[⑤]，而且还出版了很多相关的书籍[⑥]。现在已经取代 Trac，成为项目现场常用的缺陷管理系统。

---

① 就中国国内的情况来看，目前好像还没有什么活跃的社区或相关书籍。只有 Trac 的 0.12 版支持中文显示。——译者注

② http://sourceforge.jp/projects/traclight/

③ http://kanon.ultimania.org/

④ http://www.redmine.org/

⑤ http://redmine.jp/

⑥ http://redmine.jp/

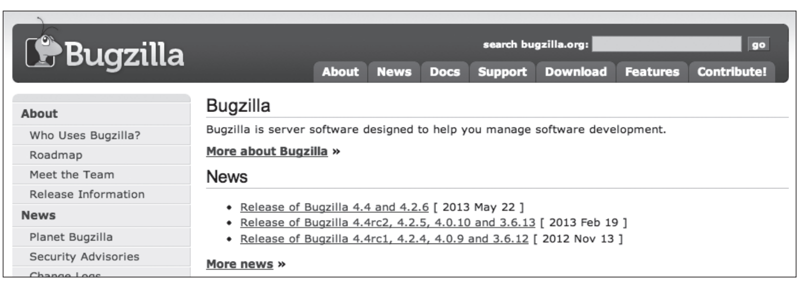
图 4.2 Redmine



●········ Bugzilla

Bugzilla[①]是基于 Perl 的缺陷管理系统（图 4.3）。原本是 Netscape 公司内部的系统，后来被作为 OSS 公开。现在由 Mozilla 基金会继续开发，可以说是 OSS 界最具“历史和传统”性质的缺陷管理系统。

图 4.3 Bugzilla



① http://www.bugzilla.org/

至今 Mozilla（Firefox）、Linux 内核、Java 的 IDE（Integrated Development Environment，集成开发环境）Eclipse[①]等历史较长的 OSS 项目都使用 Bugzilla 作为缺陷管理系统。

Bugzilla 有着数量众多的插件，是一款经得起实际使用的优秀系统。但 UI 较老，习惯了现代 Web 程序的用户可能用起来比较吃力。虽然几乎没有新项目的开发现场会选用 Bugzilla，但至今仍有相当多的现场还在使用 Bugzilla，今后也会继续使用下去。

●……… Mantis

Mantis[②]是基于 PHP 的缺陷管理系统（图 4.4）。由日本人开发，过去在日本曾被广泛使用，但近年来几乎看不到了。新项目一般不会使用 Mantis，但在参加历史较久的项目时，还是有机会看到 Mantis 的。

图 4.4 Mantis



① http://www.eclipse.org/

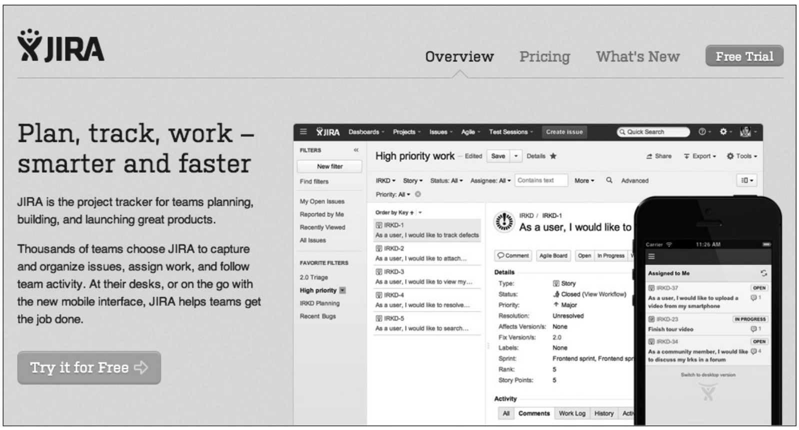
② http://www.mantisbt.org/

4.2.2　商用产品

●……… JIRA

JIRA[①] 是由 Atlassian[②]有偿提供的基于 Java 的缺陷管理系统（图 4.5）。虽说是商用的，但有 30 天的免费试用期。该产品可以以安装文件或 SaaS 的形式提供，功能丰富。如果项目预算允许的话，可以考虑使用 JIRA。

图 4.5　JIRA



从 bug 的管理到需求的管理，JIRA 能够以不同的粒度管理问题票，还可以和版本管理系统进行关联。JIRA 拥有大量的插件，能够满足各类需求。JIRA 同样支持日语[③]。但是 Wiki 功能由同属于 Atlassian 的其他产品 Confluence[④]来提供，这点需要注意。

---

① https://www.atlassian.com/software/jira/

② https://www.atlassian.com/

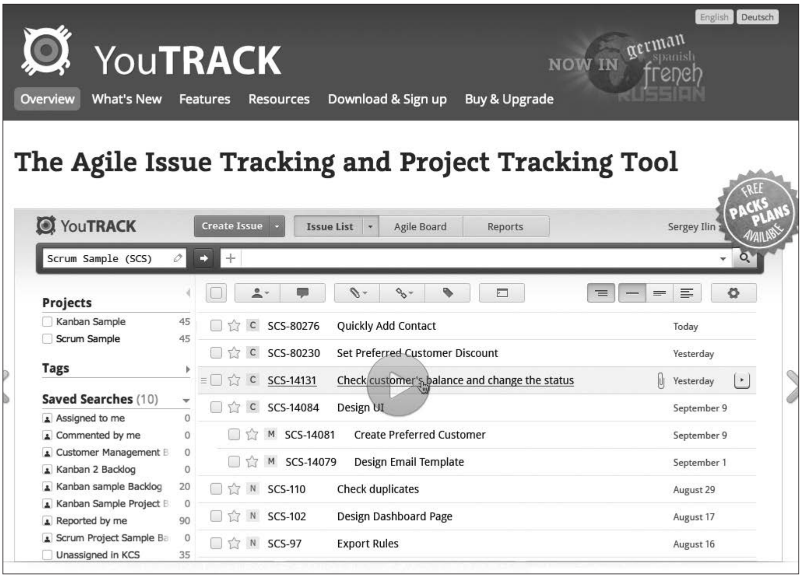
③ JIRA 本身不支持中文，但有收费的第三方汉化插件 http://www.confluence.cn/pages/viewpage.action?pageId=2164040。——译者注

④ https://www.atlassian.com/software/confluence/

●……… YouTRACK

YouTRACK[1]是以 IntelliJ IDEA 等 IDE 出名的 JerBrains[2]所提供的基于 Java 的缺陷管理系统（图 4.6）。这也是一款收费软件，但 10 个用户之内的可以免费使用。提供安装文件和 SaaS 两种形式。但需要注意的是该软件还没有中文版本。

图 4.6 YouTRACK



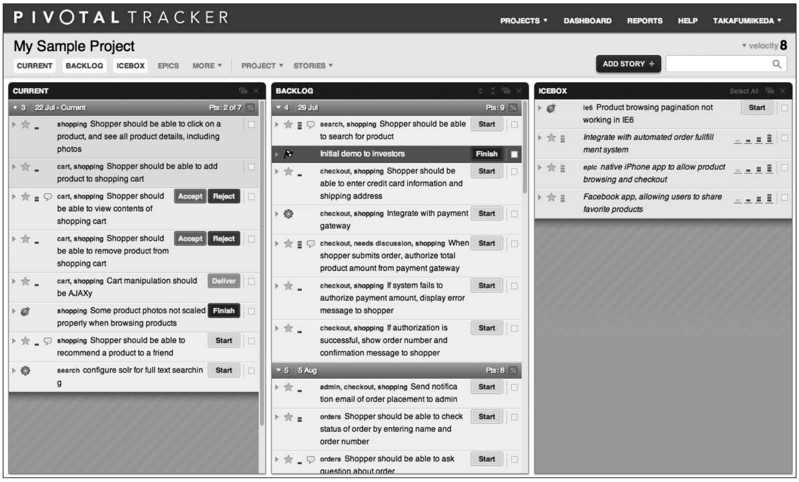
●……… Pivotal Tracker

Pivotal Tracker[3]是 PivotalLab 提供的缺陷管理系统（图 4.7）。特别适合于敏捷开发的 Scrum 开发流程。只提供 SaaS 形式的产品。

---

① http://www.jetbrains.com/youtrack/
② http://www.jetbrains.com/
③ http://www.pivotaltracker.com/

图 4.7　Pivotal Tracker



作为 Web 应用程序 Pivotal Tracker 非常有趣，并且能够通过拖曳直观地新建、移动问题票。多人同时编辑时能够在画面上实时地反映出来。UI 界面也易于使用。但是需要注意 Pivotal Tracker 没有中文版本。

如果团队成员习惯于 Scrum 的话，Pivotal Tracker 是非常方便且易用的工具。

●⋯⋯ Backlog

Backlog[①]是由日本的 nulab [②]提供的缺陷管理系统（图 4.8）。支持 SaaS 和安装文件两种形式。因为是日本公司的产品，所以对日语的支持方面完全没有问题，当然也支持多语言，包括中文[③]。

① http://www.backlog.jp/

② http://www.nulab.co.jp/

③ http://backlogtool.com/cn/

图 4.8 Backlog



Backlog 能够与 bug 管理、代码管理相关联，也支持 Wiki 的信息共享，必要的功能可谓一应俱全。10 名用户、1 个项目以内可以免费使用。

Backlog 以亲切易用的 UI 以及可爱的设计为特征，还可以使用表情符号，像使用手机一样发送表情。程序员以外的人用起来也能得心应手[①]。

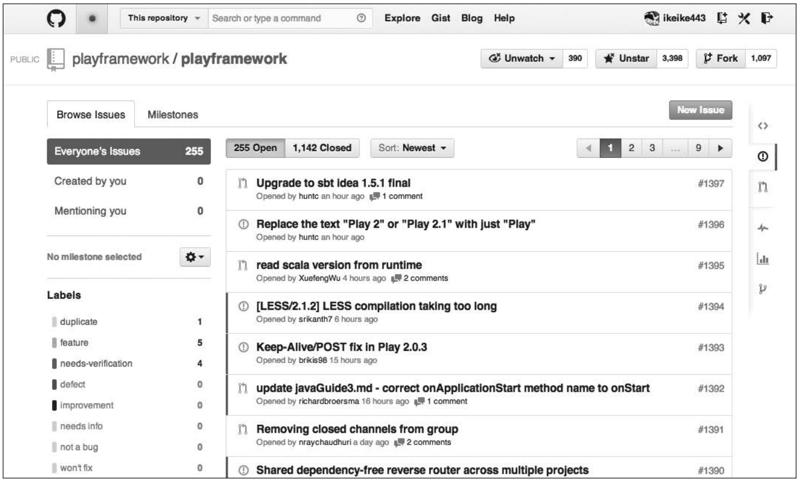
●……… GitHub

GitHub[②]已经在第 3 章介绍过了。作为缺陷管理系统，GitHub 同样具备不俗的功能（图 4.9）。检索功能等可能稍微弱了些，但和代码管理的交互做得非常出色，还具备 Wiki 的功能。并且还有名为 GitHub Pages 的用于建立项目网页的功能，以及类似于 Travis CI 的 CI 工具。可以毫不夸张地说，GitHub 能够提供 OSS 项目所需要的所有功能。

① 本书写作时就使用了 Backlog 的免费缺陷管理系统，用 Backlog 提供的 Git 来管理原稿。

② https://github.com/

图 4.9　GitHub



问题在于没有中文版本，并且基本上只有面向程序员的功能，程序员以外的项目成员使用起来可能有些困难。

GitHub 虽然可以免费使用。但如果不想让其他人看到项目的内容的话，就需要购买私有代码仓库，将项目设置为私有，这点请注意。第 3 章中已经提到过，GitHub 还提供安装文件形式的 GitHub Enterprise，在公司内部使用的话可以考虑 GitHub Enterprise。

4.2.3　选择工具（缺陷管理系统）的要点

至此我们对主要的缺陷管理系统进行了介绍。无论是 OSS 还是商用产品，数量都相当多，选择起来比较困难。并且缺陷管理系统并不像版本管理系统以及 CI 工具那样有常规的选项。

除了上面介绍的几款之外，还有很多其他的工具，大家也可以试着挑选适合自己的开发现场的工具。选择工具时需要确认的关键点因开发现场而异，但特别需要注意的是工具所提供的扩展性如何，以及是否易于实现。

缺陷管理系统和版本管理系统以及 CI 工具不同，工程师以外的项目利益相关者也会比较多地接触到缺陷管理系统。并且很多缺陷管理系

统都和团队的业务流程密切相关。因此，能否扩展缺陷管理系统，使其灵活地适应现场的开发流程，这一点同样很重要。

这里介绍的缺陷管理系统都支持以某种形式进行扩展，一般会提供 REST API 或插件机制等。

如果选用本次介绍的工具之外的其他工具，选择时一定要注意其扩展性。

专栏　缺陷管理系统的应用事例

原本用于软件开发项目中的任务管理的缺陷管理系统，近年来逐渐被扩展应用到了其他各个领域。

例如，笔者所在的团队就在下列业务中用到了缺陷管理系统。

- **用户咨询的管理**
- **委托其他部门的工作的管理**

这是因为缺陷管理系统所具有的定义任务、确定责任人和期限这样的任务进度管理功能对几乎所有的工作都是通用的。

想必本书的读者基本上都是软件开发相关的工程师，缺陷管理系统能够在各类业务中起到作用，所以请一定在软件开发项目之外也试着应用一下。

4.3 缺陷管理系统与版本管理系统的关联

随着缺陷管理系统的导入，任务的管理变得简单，项目也开始顺利运转，接着应该着手的就是和版本管理系统的关联了。关联问题票和代码的修改能够使问题或 bug 的调查变得方便，由此对项目的管理也会更为得心应手。

4.3.1 通过关联实现的功能

通过关联两个系统能够实现如下这些功能。

●……… 从提交链接到问题票

利用这个功能能够从代码的修改追溯到原本的问题票。特别是在以开发者的角度来调查代码的修改时，该功能非常方便。在缺陷管理系统和版本管理系统密切整合的系统中可以使用该功能，具体来说有 Trac、Redmine、Backlog、Github 等系统。

相反，缺陷管理系统使用 JIRA 或 Pivotal Tracker，版本管理系统使用另外的 Git（GitHub）的情况下，要在版本管理系统的提交记录中添加缺陷管理系统的问题票的链接就比较困难了，这点请注意。

●……… 从问题票链接到提交

和上述功能相反，这是一个从问题票出发，追查做出了怎样的修改的功能。主要是在从 QA 负责人、顾客或者和顾客立场相近的利益相关者的角度出发，来确认发生问题的始末、调查影响范围时，该功能非常方便。即使在缺陷管理和版本管理使用各自独立的系统的情况下，只要进行适当的配置，多数情况下还是能够使用该功能的，所以应该尽量地加以利用。

●········ 提交的同时修改问题票的状态

这是一个在向版本管理系统进行提交的同时修改相关联的问题票的状态的功能。该功能能够使开发人员的提交作业和项目的工作流程相关联，帮助加快开发速度。

4.3.2 关联的配置方法

几乎所有的缺陷管理系统都具备和版本管理系统相关联的功能，配置的方法也基本相同。实际进行配置时，只要参考文档或相关书籍等资料，应该都能够顺利地进行。

这里，我们以使用 Git 版本管理系统为前提[①]，就上述介绍的缺陷管理系统之中的一些主要产品的配置方法进行简单的说明。

4.3.3 GitHub

●········ GitHub 的 issue

关联 GitHub 自带的 issue 功能和代码修改是非常简单的，只需在提交记录中像下面这样加入 issue 的号码即可。

```
Fixed #21
```

这样从提交到 21 号 issue 的链接就建立起来了。GitHub 还会在 21 号的 issue 中添加相应的提交记录的链接，并把其状态修改为 close。可以说必要的功能是一应俱全的。

像这样，GitHub 较好地具备了关联缺陷管理系统和版本管理系统的功能。就这样运用 GitHub 应该也没有什么太大的问题，只是很多开发团队已经在使用其他缺陷管理系统。另外，因为需要使用 GitHub 所不具备的报表生成功能而另行选用其他缺陷管理系统的团队也不在少数。

在这样的情况下，可以将 GitHub 彻底地作为版本管理系统来使用，

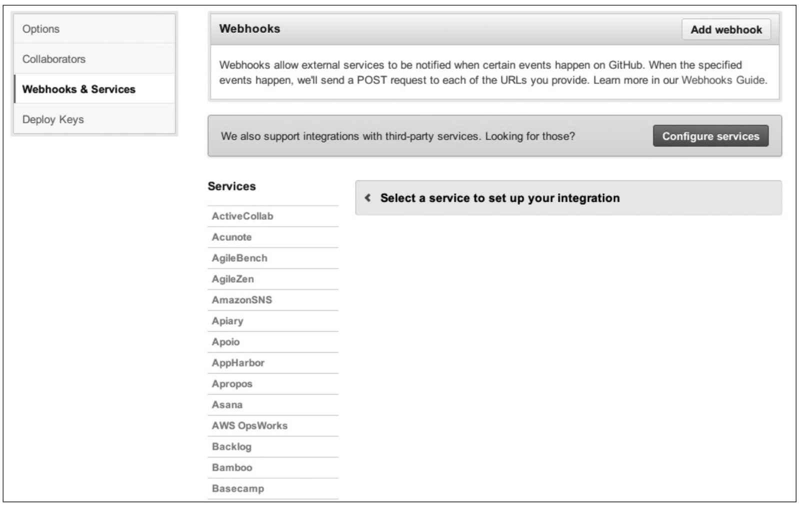
① 关于 Subversion 和缺陷管理系统的关联方法，请参考文档、书籍或网络资料等。

使用其他的产品作为缺陷管理系统。GitHub 同样提供这样的功能。

●……… Service Hooks

GitHub 为每个代码库提供了和其他服务相关联的机制（图 4.10），该机制称为 Service Hooks。它能够在向 Git Push 代码时通过 Hook（挂钩）和其他服务进行信息交互。在各代码库的菜单中依次选择 Settings>Webhooks&Services>Configure services，就可以进行确认。配置时需要该代码库的 Admin 权限，这点请注意。

图 4.10　Service Hooks



如图 4.10，GitHub 提供了世界上几乎所有系统的 Hook。从中找出自己所使用的缺陷管理系统的 Hook 并进行配置，这样就实现了 GitHub 和缺陷管理系统的关联。

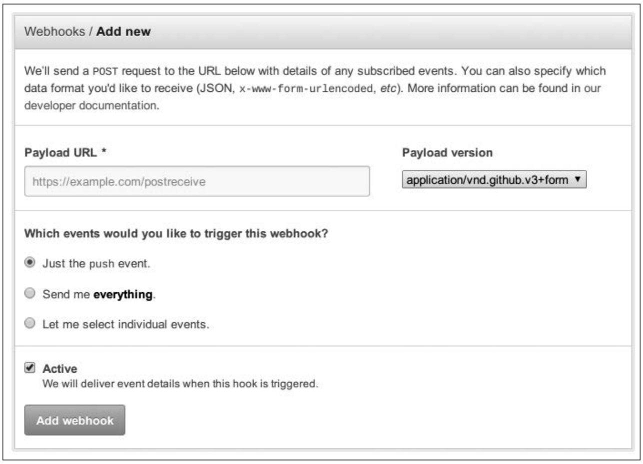
而万一没有提供相应的 Hook 的话，则有以下两个办法。

其一就是配置 Webhooks（图 4.11）。这是一种向设定的 URL 发送固定格式的 POST 请求的通用做法。在和公司内部独自开发的系统关联时就能发挥作用。顺便说一下，GitHub 发送请求的服务器的 IP 地址是公开的[1]，所以在和公司内部系统进行关联时还可以使用 IP 过滤，消除

① http://developer.github.com/v3/meta/

安全方面的隐患。

图 4.11 Webhooks



其二就是自行开发 Service Hooks。GitHub 的 Service Hooks 是作为 OSS 公开在 GitHub 上的[①]。只要按照说明开发 Hook 并公开，全世界的工程师就都能使用该 Service Hook。

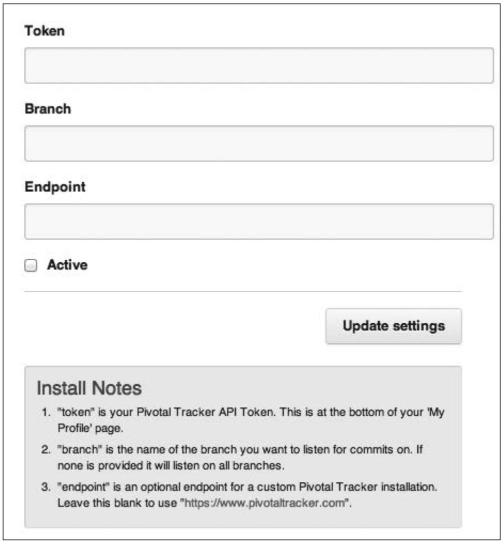
●……… GitHub 和 Pivotal Tracker 的关联

Pivotal Tracker 是 GitHub 的 Service Hooks 中提供 Hook 的服务之一。

只需要配置 Pivotal Tracker 所发行的 Token 和对象 Branch（分支）即可运行（图 4.12）。Branch 处空白的话，每次 Push 修改时都会监听所有分支，并在相应的 PivotalTracker 的问题票中添加链接。EndPoint 通常可以为空，Pivotal Tracker 的话只有在使用 custom domain 的情况下才需要配置 EndPoint。

① https://github.com/github/github-services

图 4.12　GitHub 的 Service Hook（Pivotal Trakcer）



配置完成后，在提交时添加如下提交记录，则对应的问题票（这里是问题票 ID123456）和修改的内容就关联起来了。

```
[#123456] 修正了发送邮件的bug
```

用“[]”将票号（#+ 号码）围起来。这时如果该问题票的状态是“not started”的话，则状态会自动更新为“started”。

若要在关联修改的同时把问题票的状态设置为“finished”，可以像下面这样添加提交记录。

```
[Fixes #123456] 修正了发送邮件的时机问题
```

其他的格式可以参考 Pivotal Tracker 的帮助[①]。成功关联 Pivotal Tracker 和 GitHub 的话，如图 4.13 所示，问题票和代码修改之间的关联就建立起来了。

① https://www.pivotaltracker.com/help/api?version=v3#scm_post_commit_message_syntax

图 4.13 建立问题票和代码修改之间的关联



GitHub 和 JIRA 的关联

JIRA 也和 Pivotal Tracker 一样，GitHub 在 Service Hooks 中提供了相应的 Hook。只需输入必要的信息，就能和 GitHub 进行关联。

提交时输入如下提交记录即可。JIRA 的情况下，`COOLAPP-123456` 这部分是对应的问题票号。

```
[#COOLAPP-123456] 修正了发送邮件的bug
```

使用 Service Hooks 的关联是以代码库为单位进行配置的，所以每增加 1 个代码库都必须对其进行配置。如果觉得麻烦的话可以用一些灵活的方法来应对，例如用 JIRA DVCS Connector Plugin[①]对 GitHub

① https://marketplace.atlassian.com/plugins/com.atlassian.jira.plugins.jira-bitbucket-connector-plugin

Organization 进行统一配置等。各位可以试着使用这些功能。

4.3.4 Trac/Redmine

在 Subversion 还是主流的版本管理系统的时代，比较多见的是使用 Trac 或者 Redmine 对问题票和代码进行管理。那时一般是自行在服务器上建立 Subversion 的代码库，将 Trac 或 Redmine 设置为代码库的浏览器，再配置相互之间的关联。

但自从 GitHub 出现后，Git 的简洁以及 GitHub 的 Pull Request 的便利开始逐渐为人所知并得到普及，现在已经没有太大的必要特意自行构建 Git 仓库，并使用 Trac 或 Redmine 作为代码库的浏览器了。坦率地说，如今使用 GitHub 的费用也已经比较合理了。完全可以用 GitHub 作为代码库浏览器，而将 Trac 或 Redmine 作为纯粹的缺陷管理系统来使用。

GitHub 的 Service Hooks 中已经提供了 Trac 和 Redmine 的服务，请加以有效使用。

4.3.5 Backlog

Backlog 原本是作为提供类似于 Trac 和 Redmine 的功能的 SaaS 开始开发的。其特征是支持 Subversion，和 Trac 和 Redmine 一样可以作为代码库浏览器来使用，问题票和代码修改的关联容易。

Backlog 最近也开始支持 Git 了。在线托管 Git 的 SaaS 除了 GitHub 之外并不多，日本国内比较有名的也就只有 Backlog 了。使用 Git 作为版本管理系统的话，GitHub 在大多数情况下都是最好的选择。但考虑到项目的差异以及预算，Backlog 也是非常有竞争力的选项。

并且，GitHub 的 Service Hooks 中公开有 Backlog 服务，所以可以采用这样的方法：将 Backlog 作为纯粹的缺陷管理系统来使用，版本管理系统则使用 GitHub。

●……… Backlog 和 Git 的关联

关联 Backlog 和 Git 的配置比较简单。可以分别为每一个项目进行配置（图 4.14）。只需要在各项目的“项目设置 >Git> 设置”中选中“连接课题和关键字”即可。

图 4.14 关联 Backlog 和 Git 的配置



配置完成后，在提交记录中输入如下内容即可进行关联。

```
TEAMDEV-31
缺陷管理功能和版本管理的关联方法。未完，这部分结束后前半部分结束。
```

如图 4.15 所示，问题票和代码修改关联起来了。并且，从代码到问题票的链接也如图 4.16 这样建立起来了。

图 4.15 关联问题票和代码修改



图 4.16 从代码到问题票的链接



●········ Backlog 和 GitHub 的关联

Backlog 公开在 GitHub 的 Service Hooks 中，所以和 GitHub 的关联也是非常容易的（图 4.17）。

图 4.17 GitHub 的 Service Hook（Backlog）



4.3.6 Git 自带的 Hook 的使用方法

使用像 GitHub 和 Backlog 这样的系统，缺陷管理系统和版本管理系统之间的关联的确比较容易。但因为无法定制 Git 自身，所以自由度和灵活度方面还是受到了限制。之前提到的系统大部分只公开了 Git 的服务器端 Hook 中的 Post-receive Hook，并没有公开所有的 Hook。

自己搭建 Git 服务器的话，就能够使用 Git 所提供的所有 Hook。在 Git 的代码库的 .git/hooks/ 目录下，Git 提供了许多分别运行在服务器端以及客户端的 Hook。该目录下存放有默认文件名为 *.sample 的示例脚本，以此文件为样本进行开发，可以写出各种各样的 Hook[①]。

① 详细内容请参考 Git 的文档。http://git-scm.com/book/zh/ 自定义 -Git-Git 挂钩。

4.4 新功能开发、修改 bug 时的工作流程

缺陷管理系统和版本管理系统关联后，新功能开发 / 修改 bug 时的工作流程如下。本节以 JIRA 缺陷管理系统、Git（GitHub）版本管理系统为例来进行说明。

❶ 建立问题票
❷ 指定责任人
❸ 开发
❹ 提交
❺ Push 到代码库

4.4.1 工作流程

❶ 建立问题票

首先根据问题内容建立问题票（图 4.18）。

图 4.18　建立问题票



❷ 指定负责人

指定负责该问题票的开发人员[①]。

❸ 开发

开发人员将问题票的状态更改为"In Progress"（作业中）并开始作业。

❹ 提交

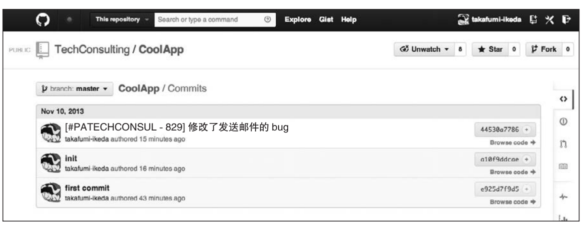
为了实现或修改功能，对代码进行修改并提交。这时要在提交记录中添加问题票的票号。不同缺陷管理系统对应的提交记录格式可能有所不同，JIRA 的提交记录如下。

```
[#PATECHCONSUL - 829] 修改了发送邮件的bug
```

❺ Push 到代码库

在将提交的内容 Push 到代码库（图 4.19）的同时，系统会在关联的问题票中添加提交的链接（图 4.20）。这样问题票和提交之间的关联就建立起来了[②]。

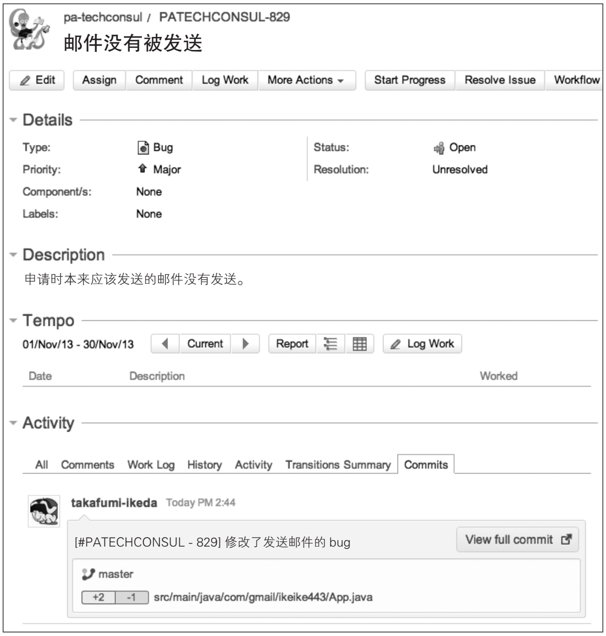
图 4.19　代码库的提交记录



① 在传统的项目管理中，由项目经理来指派合适的开发人员。而在敏捷开发的团队中，则是开发人员自行建立问题票，再将问题票指派给自己。特别是采用 Scrum 的团队开发，问题票建立时负责人一栏是空白的，在每天的例会（晨会）上由开发人员自己举手认领问题票。

② 在这个例子中没有对提交时的提交记录进行检查，因此即使忘记在提交记录中添加问题票号也能够顺利提交。如果希望系统强制要求输入问题票号的话，可以使用版本管理系统提供的 Hook 机制，在提交时对是否记载了问题票号进行检查。

图 4.20 在问题票中添加链接



在介绍 Pivotal Tracker 和 Backlog 时已经提到过，在提交记录的问题票号中插入 `Fixs`、`Fixed`、`Delivered`等特定的命令语句的话，在添加链接的同时还能改变问题票的状态。命令语句的格式以及与之对应的状态变化，各个缺陷管理系统不尽相同，请参考相应的帮助文档。

4.5 回答“那个 bug 是什么时候修正的”的问题

团队开发过程中经常会从销售、顾问或者客户那里收到“那时的那个 bug，是什么时候、在哪个版本修正的?”这样的调查要求。

这时，如果已经关联了缺陷管理系统和版本管理系统的话，就有可能顺利地找出答案。

4.5.1 Pivotal Tracker 的例子

首先以 Pivotal Tracker 为例进行说明。

●……… 用记忆中残留的关键字进行检索

如果在收到这样的调查要求时能够得知问题票号，那就不必再去查找问题票了。但如果无法获得，那就只能通过调查内容中模糊的信息以及负责人模糊的记忆来找出问题票。

●……… 检索

首先使用缺陷管理系统的检索功能找出问题票（图 4.21）。这里用关键字“一并下载”进行全文检索。

图 4.21 用 Pivotal Tracker 进行关键字检索



●……… 通过问题票查找代码修改

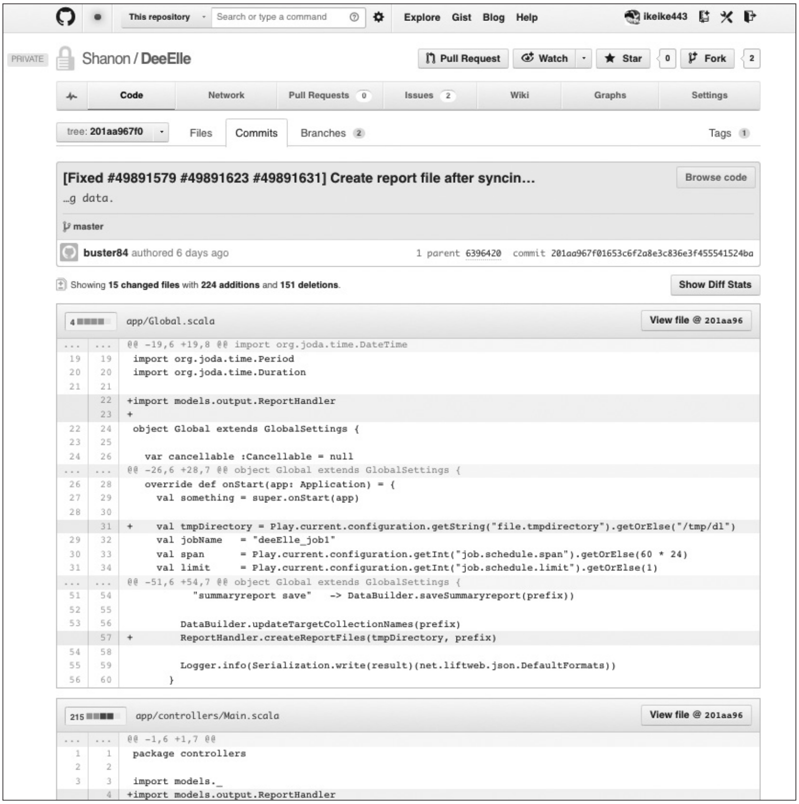
找到的问题票中记载有指向 GitHub 修改记录的链接（图 4.22）。

图 4.22 修改记录的链接



点击链接就会跳转到 GitHub 对应的提交页面，在此可以查看提交代码的差分对比（Diff）（图 4.23）。

图 4.23 GitHub 的修改记录



这样就查到了对应的代码提交纪录。接着只要查出这个提交反映到了哪个发布版本[①]，调查就基本可以结束了。

根据运用方式的不同，例如在 master 分支适当地打上发布标签并进行发布的情况下，只需要确认提交是否反映到 master 分支，就能够判断是否反映到某个版本的发布中了。

4.5.2 Backlog 的例子

再举一个 Backlog 的例子。这次版本管理系统使用 Subversion。

① 通常以标签的形式保存。

●……… 检索

在 Backlog 的画面上设置检索条件，对问题票进行检索（图 4.24）。这里以“程序令牌”作为关键字来检索。

图 4.24　检索问题票



确认找到的问题票的内容（图 4.25）。

图 4.25　问题票的内容



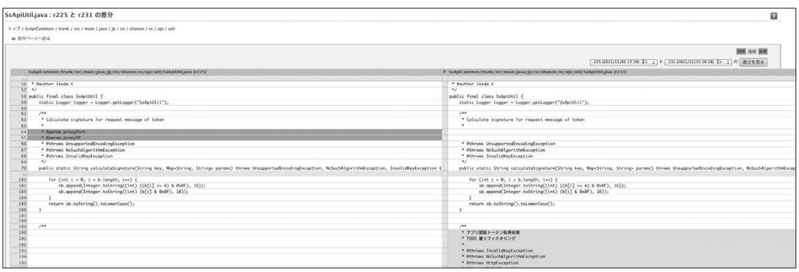
问题票中有提交记录的链接，点击链接就能确认提交的内容（图 4.26）。

图 4.26　提交记录列表



还可以通过 Diff 来比较代码的内容（图 4.27）。

图 4.27　比较代码内容



这样就通过 Backlog 将从调查到代码的修改这一整个过程串联起来了。接着也同样只需要调查清楚这些提交反映到了哪个版本的发布中就可以了。

4.6 回答“为什么要这样修改”的问题

在 4.5 节中，我们说明了根据调查查找代码修改的例子。而在团队开发现场还存在相反的、从代码追溯问题票的情况。例如在本地编译失败或者单元测试失败的情况下，就需要通过缺陷管理系统和版本管理系统的提交记录，追查出造成失败的问题票。

但是这个功能的前提条件是必须由单个系统统一提供缺陷管理和版本管理。例如 Backlog 会自动地同步生成从提交记录到问题票的链接，能够简单地从代码修改链接到问题票。从图 4.26 中可以看出，提交记录也添加有问题票的链接。

Trac 和 Redmine 基本与 Backlog 一样。GitHub 会在提交记录中自动添加自身的 Issue 功能的链接。上述以外的缺陷管理系统基本上都无法自动添加链接。在这样的情况下，可以采用复制提交记录的版本号并粘贴到所使用的缺陷管理系统中等方法来应对。虽然稍显原始，但比起没有任何信息、完全无从调查的状况来说还是要好很多。

4.7 本章总结

综上所述，通过关联缺陷管理系统和版本管理系统，代码的可追溯性有了很大的提高。

本章介绍了包括免费 / 收费在内的各类缺陷管理系统。最近几乎所有的缺陷管理系统都提供了和版本管理系统进行关联的功能，这里对关联的方法也进行了说明。

通过关联缺陷管理系统和版本管理系统并加以有效地利用，就能够知道什么时候、谁、进行了什么操作这一连串的信息，并且还能加快发生问题时查找原因的速度。

这和既不使用缺陷管理系统也不和版本管理系统进行关联的情况比起来可以说是天壤之别。请回忆一下第 2 章中的案例分析。在那个案例中，收到通过邮件发来的调查请求，却无法检索版本管理系统的提交记录，即便确认了提交记录也不知道这样修改的理由。

第 2 章的案例分析中还有个例子是：根据收到的调查请求进行的修改被其他开发人员用别的提交覆盖掉了。如果关联了缺陷管理系统和版本管理系统的话，就能从提交记录找出对应的问题票，进而就应该能知道进行如此修改的缘由。

但即使这样还是觉得有所不足，并不是说仅仅找出有问题的提交，并调查出作为提交原因的问题票就足够了。不能一开始就阻止这样有问题的提交吗？不能在每次提交时都对其正确性进行检查吗？

如果能够实现上述机制，就能在更早的阶段确保提交的正确性，开发的质量也会相应地提高。这样的机制就是将在第 5 章进行说明的 CI。

专栏　缺陷管理、bug 管理以及需求管理

本章主要讲述了用缺陷管理系统对开发中的任务进行管理的相关内容。而在任务确立之前的阶段，软件开发团队还有必要对课题

进行管理。

缺陷管理系统能够覆盖哪些范围呢？或者说应该覆盖哪些范围呢？

这里我们借鉴敏捷开发方法论之一的 Scrum 的部分术语，将课题分为 epic、story、task、bug 这 4 类，并分别对它们的粒度进行说明。

epic（很大的需求、跨部门的课题等）

可以称为 epic 的大粒度的课题一般有如下特征。不熟悉 epic 这个术语的话可以回想一下系统需求确定下来之前的需求定义阶段，这样会比较容易理解。

- **和工程师不相关的事情占多数**
- **以确定优先顺序为主要目的，因此以一览表的形式呈现出来很重要**
- **刚开始时多处于模糊的状态，因此不适合采用固定的格式，最好是能够灵活修改的**
- **方便多人进行编辑、共享**

这样的粒度下一般会使用电子表格工具。因为工程师之外的团队成员也能够直观地进行编辑，所以是最简便的方法。如果考虑到共享和同时编辑的问题，最好使用 GoogleDocs 这样的在线工具。

epic 的内容示例：

- **构筑开拓新市场用的系统**
- **为每个客户准备信息中心，提供对其所使用的产品的统计信息和用户的访问情况进行分析的功能**

story（具体的需求，和功能性需求或规格程度相近）

称为 stroy 的粒度有如下这些特征。它在 Scrum 中表示为用户带来的价值。如果对 Scrum 不熟悉的话可以理解为程度上近似于具体的功能要求或规格。

- **一般多由产品的负责人编写（Scrum 中的用语是 product owner）**
- **根据团队或项目的情况，可以由工程师以外的成员编写，也可以由程序员编写**

- **需要进行状态管理**
- **最好开始和代码建立关联**
- **负责人可以是多人也可以是一个人**

这样的粒度可以考虑使用缺陷管理系统进行管理。关系到多个负责人的大型 story 的话需要再想想办法。

方法之一就是先建立问题票再分割为子任务。如果系统支持在问题票之间建立继承关系，可以直接使用该功能，这样管理起来也较为方便。

根据相关人员的数量，也可能是使用电子表格工具进行管理更为理想。至于应该如何确定缺陷管理系统和电子表格工具的分界线，每个项目、每个团队都有自己不同的标准，很难说哪个是最好的，请根据项目的实际情况探讨合适的方法。

比较常用的方法是以团队为分界线。如果课题涉及多个部门，可以采用电子表格工具进行管理，等具体的开发内容确定下来后再由各部门自行建立问题票。由于管理方面职责明确，因此较大的开发团队多采用该方法。

stroy 的内容示例：

- **为了体现出对顾客的欢迎，想要在顾客登录时根据顾客的信息显示欢迎或其他消息等**
- **将每个用户 ID 的用户访问状况，以小时、属性分别进行统计，并且支持在信息中心阅览和下载**

task

即作业。和价值没有直接关系，但如果不完成 task 的话 story 的完成就无从谈起。task 的概念应该还是比较容易理解的。

- **具体的作业内容**
- **大致的印象是 story 下关联着多个 task**
- **只有一个负责人**
- **根据内容确定是否需要和代码进行关联**

task 应该采用缺陷管理系统进行管理。如果系统支持问题票之

间有继承关系，或者允许在问题票下定义附属问题票，可以将 story 定义为问题票，将 task 定义为子问题票或者附属问题票。

task 的内容示例：

- **初始化系统**
- **写单元测试**

bug、故障处理、调查

即 QA 部门发来的 bug 报告、故障的处理以及顾客的咨询等。

- **程度上和 task 相近**
- **通常由单个开发人员负责**
- **需要和代码进行关联**

应该作为缺陷管理系统的管理对象。至于是以 task 的粒度进行管理还是以 story 的粒度进行管理，可以根据项目或团队的具体情况选择合适的方式。

在 bug 数量很多的情况下，可以将 bug 管理和其他的问题票区分开，使用其他的系统进行管理。这也是一种方法，但这样的话管理上就会变得复杂，所以并不推荐。应该尽可能地在同一缺陷管理系统下对 story 和 bug 进行管理。

总结

至此我们一起看了课题的分类和管理方法。这样的分类方法是灵活的，并非一成不变的。如果团队有自己一贯采用的分类方法也完全适用。

能够肯定的是：缺陷管理系统适用于目的以及内容在一定程度上确定下来之后的管理，不适合在此之前的那些比较模糊的、不确定的事物的管理。在这个阶段，使用电子表格工具等形式不固定的工具进行管理会比较好。

表格工具和缺陷管理系统的分界线取决于相关的部门数或人数，另外，在负责的部门发生变化时切换到缺陷管理系统也是可以的。

没有必要只使用一种工具，同样，也没有必要拘泥于一种定义，我们要做的应该是摸索出一套最适合自己团队的方法。

第 5 章

CI（持续集成）

5.1 CI（持续集成）

5.1.1 什么是 CI（持续集成）

团队开发中最重要的工作是什么？是编码等开发工作？还是测试工作？当然这些工作都是团队开发中基本且重要的工作。但是为了顺利地推进由多人参与的开发以及测试，最重要的基础是“集成”。

●……集成（integration）

将各个成员的工作成果集中到一处进行集成，直到形成可以运行的系统，各个成员的开发工作才有了意义，才能进行测试，最终才能以产品的形式向顾客提供价值。

集成，具体来说就是像下面这样执行 build 和测试的流程。

- **将所有的代码集中到一处**
- **设置依赖程序库等的路径**
- **必要的情况下进行编译**
- **进行数据库构建和数据加载**
- **必要的情况下对中间件进行配置和启动**
- **实施单元测试、集成测试、用户验收测试等**

●……持续地进行集成就是 CI

将这些集成处理流程持续地执行，就是 CI（Continuous Integration，持续集成）[①]。

① 如果从一开始就能实现之前提到的所有集成处理流程的自动化，这个自然是最理想的。但这将耗费大量的时间和精力，因此没有必要将实现所有集成处理的自动化作为目标。不仅仅是 CI，类似的实践的效果以及对应的开销都需要进行权衡。因此，可以先从自己项目所必要的处理开始实现自动化。

原本 CI 是作为敏捷开发方法之一的极限编程（eXtreme Programming，XP）的实践项目被提出的。其实可以毫不过分地说，在以高品质、快速地提供有价值的软件为目标的敏捷开发中，CI 是最基本、最重要的实践。

在以前的瀑布模型开发[①]中，一般要等到所有的开发工作都结束后，到测试阶段时才开始着手进行集成。

想必读者都有过这样的经验：直到测试阶段才开始实施集成，往往会出现程序库缺失、数据库构建失败、编译无法通过等问题。集成的实施越迟、代码的量越大，就越困难。

要集成多名团队成员的作业内容更是一件相当麻烦的工作，因此往往容易往后拖延。但是只有经过了集成这个步骤，程序才能作为一个整体开始运行。在进行集成之前，确认产品是否满足需求，是否有奇怪的 bug 等检证工作也无法开展。可见拖延并不是一个好办法。

不将复杂的集成处理推后，而是在开发中时常地进行集成处理，由此来排除软件开发中的复杂性，这就是 CI 的思考方式。

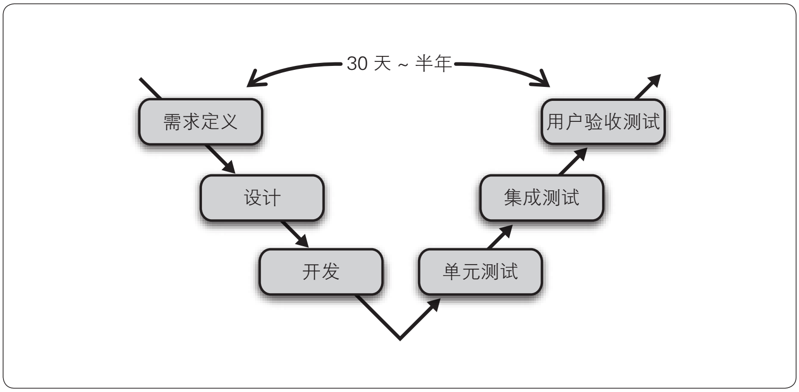
5.1.2 使开发敏捷化

●瀑布式开发的开发阶段

过去的瀑布式开发是在需求定义完成后进行设计、开发，之后再依次进行单元测试、集成测试、用户验收测试，因此到开始正式提供服务的开发周期容易拖得很长（图 5.1）。

① 将系统的开发流程分为分析、设计、编码、测试、部署等阶段，并按上述顺序依次进行作业的开发模型。

图 5.1 瀑布式开发的开发阶段



开发全部完成后才开始单元测试，这就意味着如果开发周期为 3 个月，那么多名开发人员为期 3 个月所编写的代码直到单元测试阶段才第一次集中到一处进行 build。在此之后还有集成测试和用户验收测试。

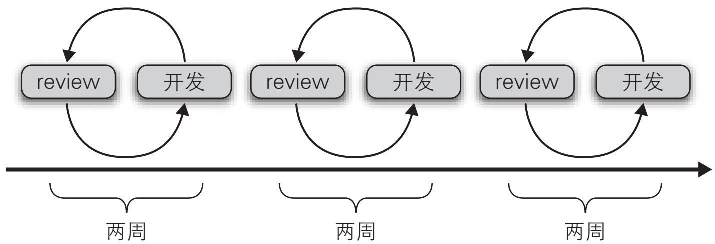
从图 5.1 就能看出，从需求定义确定下来，到用可以实际运行的程序进行验证，这期间的时间间隔非常大。甚至会有在项目临近结束时还需要重新返工的风险。

瀑布式开发是确保各阶段的正确性之后再开始下一阶段，因此理论上在进入下一个阶段后就不应该返回上一阶段。但实际上，很多事情只有在目睹运行的系统后才会知道。正如读者所知，很多采用瀑布式开发的项目直到临近结束还在修改需求、重新编码并重新测试。

●……… 敏捷开发的开发阶段

与之相对，以 Scrum 为代表的敏捷开发采取了相反的方式，即以 sprint 这样较短的周期来循环实施设计开发、单元测试、集成测试、用户验收测试（图 5.2）。sprint 的周期因项目而异，通常多设定为两周到 1 个月。

图 5.2　敏捷开发的开发阶段



Scrum 以 sprint 为周期提交工作成果，并以 sprint review 这样的流程对工作成果进行审查，给出反馈，并反映到下一个 sprint 的作业中。以正式提供服务为目标[①]，重复上述过程。上述流程可以理解成将瀑布式开发中从需求定义到用户验收测试的过程浓缩到为期两周的 sprint 内进行[②]。这样的方式能够及早地检查出产品和需求之间的差异，调整的机会也会相应增加。

CI 是敏捷开发的基础中最重要的实践。sprint 中的工作成果通常是指“可以确认满足某种需求的可运行的程序”，进一步说就是“可以判断是否能够正式投入运营的可运行程序”。要在短短两周的时间内制作出可以运行的产品，如果还要在集成上花费时间是肯定来不及的。更何况是到了 sprint 的后期才开始集成并测试，无论如何都是来不及的。因此就需要每天进行开发、集成、测试这样的循环，这就是 CI。换言之，build 和测试的自动化途径将是至关重要的。

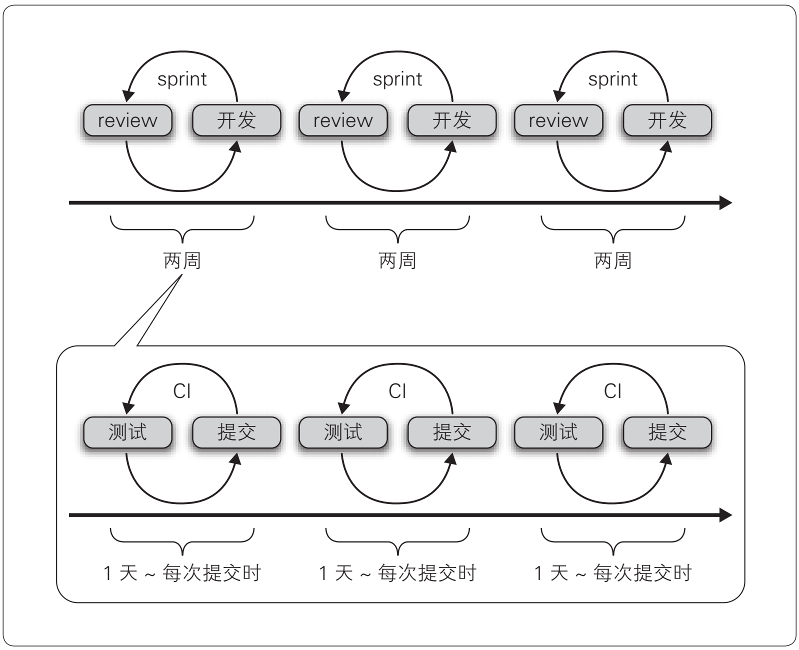
① 进一步来说，上线运营与其说是终点，倒不如说是开始。如果能在上线后持续进行 sprint，反复地改善和审查，就可以向顾客提供更多的价值。

② 当然要在两周内实现所有的需求是不可能的。因此要排出需求的优先顺序并估计工数、制定计划。从优先度高的项目开始，按顺序投入到 sprint 开发中。在每次的 sprint review 中检查开发状况，并根据市场的变化，一边调整优先度一边进入下一个开发周期，这就是敏捷开发。这里的估计工数和制定计划是敏捷开发中最困难、同时也是最有趣的部分。本书就不详述了，有兴趣的读者可以参考下面的文献资料。
《敏捷估计与规划》，（美）Mike Cohn 著，宋锐译，清华大学出版社，2007

无论是从自动化集成的角度还是从其他角度来看，CI 都是非常重要的。之前已经提到过，Scrum 以 sprint 周期为粒度反复进行 review 和反馈。实现 CI 的话，编程作业和自动化测试的循环就能粒度更细、运转更迅速。自动测试的结果正是程序是否正确运行的反馈。

也就是说，如图 5.3 所示，项目整体以 sprint 为粒度进行反馈，并这样不断循环。而每个 sprint 中又进一步以 CI 为粒度进行反馈，同样进行着循环。这样就构成了嵌套的结构图。

图 5.3　sprint 和 CI 的循环



以这样的粒度快速进行反馈的循环，就能够在快速开发的同时确保产品的质量，这就是敏捷开发。而在背后支撑着敏捷开发的就是 CI 这样的实践。

5.1.3 为什么要进行 CI 这样的实践

那么为何 CI 和敏捷开发近年来逐渐走红[①]了呢？想知道答案的话就需要从以下两个角度出发来思考。

- 成本效益（cost benefit）
- 市场变化的速度

●……… 成本效益

一般而言，bug 出现的时间越长，修改该 bug 的成本就越高。例如使用刚刚编写的代码当场进行测试，即使发现了 bug，查明原因并进行修改也是比较容易的。

与之相对，如果不进行测试直接提交代码，3 天后才注意到 bug 的话会怎么样呢？两周后、1 个月之后会怎么样呢？那时所写的代码的内容可能已经记不太清了，其他开发人员也可能对该代码进行过提交了，修改 bug 的难度将大大提升。而如果是 3 个月后或半年后才发现，想想就觉得非常恐怖了。

实现 build 和测试的自动化，并且能够实施 CI 的话就能解决这个问题。通过实施 CI，提交之后就能立即察觉是否有 bug 产生，修改 bug 的成本将大大降低。所以说 CI 的实施能大大提高成本效益。

这里所说的成本主要是指修改 bug 相关的经济方面的成本，除此之外还有其他需要注意的成本，例如可维护性相关的成本。

保持代码清晰易读、添加功能方便，这是长期维护产品中非常重要的。不仅是单纯地加快增加新功能的速度所带来的经济方面的优点，在保持负责维护和功能开发的团队成员的精神健康以及提高开发效率方面

---

① 这里虽然是说近年来才逐渐走红，事实上 CI 这样的思考方式很早之前就有了。比如《程序员修炼之道》（Andrew Hunt、David Thomas 著，马维达译，电子工业出版社，2011 年）中就提到了自动化测试的必要性；*Joel on Software*（Avram Joel Spolsky，APress，2004 年）中也以专栏的形式介绍了自动化 build 和自动化测试对提高工作效率的作用。近年来随着相应工具的出现，CI 也开始逐渐普及了。

都会起到很大的作用。

●········ 市场变化的速度

如今的市场瞬息万变。特别是网站和智能手机的 App，1 款产品的开发周期一般不足 1 至 2 年。因为在开发期间市场趋势很可能会发生变化，导致开发出来的产品失去作用。还有 Web 之外的一些类似于财务系统这样的业务系统，还会受到突然的政策修改等外部环境变化的影响。

如果每次都按照瀑布式的开发流程，从头开始全部实施的话，可以说很难跟上市场的变化。话虽如此，如果只是单纯地提早发布日期、缩短开发日程，而不在 CI 等方面下工夫的话，那只会造成 bug 和退化频发，进而导致发布的产品质量下降。效果恰恰相反。

为了应对市场的变化，在一定程度上不得不牺牲代码的可读性和可维护性。特别是在产品开发初期，这是常有的事情。不同的产品可能有所差异，一般而言投入市场越迟，就越容易失去机会。最差的情况下，辛苦开发的产品可能完全没有被用户接受的机会，产品开发的投入完全打了水漂。

●········ 兼顾开发速度和质量

如何才能既保持能够应对市场变化的开发速度，又保证高质量呢？CI 就能起到重要的作用。

例如，编写能确保产品 API 正常运行的最低限度的测试代码并执行 CI，在保证代码内容正常运行的基础上优先向市场进行发布。这个阶段代码的可读性和可维护性较差，功能添加也不方便，只是姑且能够运行并向用户提供价值而已。

确认产品能够正常使用之后，再用考虑了之前牺牲的可读性和可维护性的代码来替换原来的代码。这是为了应对下一个阶段，即进一步添加功能所必需的。

也就是说，初期优先速度，致力于尽快上市，之后再对代码进行重构以提高可维护性。而支持上述方式的正是 CI，有了 CI 上述方式才成为可能。既应对了市场的快速变化，又控制了确保可维护性和开发人员

精神健康所需的成本。

综上，从成本效益和市场的变化速度这两个角度来看，CI 这样的实践是非常重要的。

5.1.4 CI 的必要条件

“CI 的字面意思已经理解了，想开始着手实施，应该怎么做呢？从何处着手呢？”为了回答这样的问题，我们来说一下 CI 的必要条件。

开始实施 CI 的必要条件如下。

- 版本管理系统
- build 工具
- 测试代码
- CI 工具

版本管理系统

实施 CI 过程中最重要的部分就是版本管理系统。正如第 3 章中所讲解的那样，构建程序所必需的资源应该尽可能地由版本管理系统进行统一的管理，这是非常重要的。例如代码、依赖关系、数据库模式、配置文件等。使用版本管理系统，任何人、任何时候都能够获取最新的资源是非常重要的。

build 工具

同样重要的还有 build 工具。例如 make，写好 build 的定义后，只要执行一条命令，就能够进行代码编译、数据库构建和测试，并最终生成可以运行的程序。主要的 build 工具如表 5.1 所示。

表 5.1 主要的 build 工具

build 工具	说明
make	经典的 build 工具。根据 Makefile 这样的配置文件进行 build。UNIX 下的软件 build 所必不可少的工具，负责搭建服务器的工程师每天都会接触

（续）

build 工具	说明
SCons	用 Python 编写的替代 make 的工具。使用由 Python 写的配置文件 SConstruct 进行 build。支持各种语言的 build，在 OSS 界有着一定的占有率。比较有名的是 V8[①]的 build 就使用了该工具
Ant	Java 写的 build 工具，使用名为 build.xml 的 XML 文件来定义 build 顺序。因为要定义 build 顺序，所以使用起来比 Maven 稍显复杂，相反也正因为可以定义 build 顺序，所以非常灵活，在 Java 的世界中被广泛使用
Maven	由 Java 编写的项目管理工具。用 pom.xml 文件来定义 build。Maven 是根据 CoC（Convention over Configuration，惯例优先原则）所制作的工具，因此只要符合 Maven 的规则，就很少需要对 build 进行额外的定义。反之，和 Ant 相比灵活性要差了些。另外，Maven 作为项目管理工具，除了 build 以外还具有项目网站的生成、测试的执行以及部署等各类功能
Gradle	Gradle 是近年来 Java 界比较受关注的新 build 工具。它并非使用 Java，而是用名为 Groovy 的 JVM（Java Virtual Machine）上运行的脚本语言编写的。配置也可以使用 Groovy 编写，比 XML 可读性强、自由度高。还可以只使用 Gradle 进行依赖关系的管理[②]。如果觉得 Ant 过于复杂，Maven 又缺乏灵活性，可以考虑使用 Gradle 作为解决这两个问题的 build 工具
Rake	Ruby 编写的 build 工具。使用由 Ruby 编写的配置文件 Rakefile。随着 Ruby on Rails 的普及，以及以 Chef 和 Capistrano 为代表的服务器构建自动化工具[③]的普及，Rake 开始被广泛使用

当然，如果无论如何现有的工具都不可用的话，也可以用 shell 脚本或其他熟悉的语言自行编写 build 脚本，但这里并不推荐这样做。多数情况下使用表 5.1 中的 build 工具都应该能够满足需求。并且如果使用的是全栈式（full stack）Web 应用程序框架的话，有的框架也会自带 build 工具。

无论使用哪一款工具，重要的都是要具备无论谁在什么时候进行了提交，build 所需的处理都能自动进行这样的机制。

① Google 的 JavaScript 引擎。

② 实际上其内部包含了 Ivy，由 Ivy 进行依赖关系的管理。

③ Chef 和 Capistrano 都是由 Ruby 编写的工具，和同样可以用 Ruby 来定义 build 的 Rake 配合度较好。因此多数的 Chef 和 Capistrano 的任务都是由 Rake 来编写的。

测试代码

CI 中的 build 工作并不是仅仅将代码编译一下就结束了。执行测试，持续地确认应用程序的正确性也是非常重要的。通过测试来确保程序的正确性就不必担心退化的发生，能够大胆地添加新功能和进行代码重构，还能够加快开发速度、提高产品质量。用于编写测试代码的框架的相关内容将稍后讲解。

CI 工具

当然，执行 CI 需要相应的工具。CI 工具是将版本管理系统和代码、build 工具组合起来，持续地进行集成作业的工具。如果仅仅是执行自动 build、自动测试的话，不使用 CI 工具也可以进行。但通过利用 CI 工具的各类功能，能够解决很多问题，比如自动化测试何时进行，执行的结果如何显示和通知，build 结果和版本管理系统、缺陷管理系统之间的可追溯性如何确保等。

5.1.5 编写测试代码所需的框架

编写测试代码所需的框架有以下两种。

- 测试驱动开发（TDD）的框架
- 行为驱动开发（BDD）的框架

测试驱动开发（TDD）的框架

即实现测试驱动开发（Test Driven Development，TDD）所需的框架。

在编写应用程序的代码之前，为了确认需求先编写测试代码（test first），然后再编写符合测试代码的应用程序代码，这样的手法就是 TDD。这也是极限编程（XP）实践的一种。具有代表性的测试驱动开发框架有 Java 的 JUnit[①]和 TestNG [②]等，还有为数众多的各类语言的 xUnit

① http://junit.org/

② http://testng.org/

系列框架。

例如 JUnit 的情况下，对于 Sample类的 getName()函数，可以像下面这样编写测试代码。

```
public class SampleTest {
  @Test
  public final void testSamplegetNameIsTakafumi() {
    Sample sample = new Sample();
    assertThat(sample.getName(), is("Takafumi"));
  }
}
```

这些 xUnit 系列框架的特征是：通过编写测试用例，确认测试对象类以及方法的动作的正确性。换言之，也可以理解为根据测试代码来设计类的 API。这个特征和接下来的行为驱动开发工具形成了很好的对照。

●……… 行为驱动开发（BDD）的框架

从测试驱动开发发展而来，近年来逐渐流行起来的便是行为驱动开发（Behavior Driven Development，BDD）。BDD 和 TDD 一样都是最先编写测试代码。

和 TDD 的不同之处在于，TDD 是针对程序的 API 编写测试，而 BDD 则是接近于需求说明的编写方法。如同其名字一样，BDD 着眼于程序的行为。其方式是在需求确定后编写应用的代码，所以与 TDD 的测试优先相对，BDD 可以说是需求优先（spec first）。

BDD 的代表性框架有 Ruby 的 RSpec[①]和 Cucumber[②]、JavaScript 的 Jasmine[③]、Scala 的 Specs2[④]等。其他还有各个语言所对应的 xSpec 系列的框架。另外，Cucumber 方面，对 Ruby 之外的各语言的支持也在正式发布中。

例如 Cucumber 面向 Java 实现的 Cucumber-jvm 中，在名为 sample.

① http://rspec.info/
② http://cukes.info/
③ http://jasmine.github.io/
④ http://specs2.org/

feature 的文件中用自然语言如下这样描述需求。

```
#language: ja
 特征（feature）: Sample功能
   场景（scenario）: 确认Sample类中的Name属性
   前提: new Sample类
   结果: getName的返回值应该为“Takafumi”
```

接着如下定义空的测试类。

```
import org.junit.runner.RunWith;
import cucumber.junit.Cucumber;

@RunWith(Cucumber.class)
@Cucumber.Options(format={"pretty"})
public class SampleTest {
}
```

定义好之后，像通常的 JUnit 一样执行上述文件，标准输出会显示如下代码。

```
You can implement missing steps with the snippets below:

@前提("^: new Sample类$")
public void new _Sample类() throws Throwable {
    // Express the Regexp above with the code you wish you had
    throw new PendingException();
}

@结果("^: getName的返回值应该为\"([^\"]*)\"$")
public void _getName应该为(String arg_name) throws Throwable {
    // Express the Regexp above with the code you wish you had
    throw new PendingException();
}
```

这就是用自然语言描述的 sample.feature 所对应的测试代码的雏形。将这段雏形代码复制粘贴到刚才空的测试类中并加以实现。本例的具体实现如下。

之后只需要向 sample.feature 用自然语言添加需求并执行测试即可。向 sample.feature 中添加的需求如果没有在测试类中进行实现的话，就会在标准输出中显示代码，提示要添加新的测试函数。

像这样根据自然语言描述的需求（行为的定义）来编写测试代码，这就是 Cucumber 的方式。

这些处理在技术上和 JUnit 并没有太大的变化，区别仅在于测试的编写方式和执行方式。虽说如此，也正是因为这些不同之处，Cucumber 才使验收人员能够用自然语言编写需求规格。

这次用 Cucumber 试着写了 Sample 类的单元测试。本来 Cucumber 是用来定义行为并进行测试的工具，所以一般多用于结合多个类的测试。也就是说，Cucumber 原本是用于确保集成测试和用户验收测试的。这次是为了对比 TDD 框架和 BDD 框架而特意编写了相同的测试。

大多数的需求都能用接近自然语言的 DSL（Domain Specific Language）[①]来编写，这是 BDD 的特点。使用近似于自然语言的 DSL 来描述需求，然后直接将其用于测试。BDD 是以让负责需求定义的人或客户等立场上接近于产品利益相关者的人也能够编写需求为目的而设计的，这也是其特征。与之相对，TDD 框架的机制是促进 API 的设计。

5.1.6 主要的 CI 工具

CI 工具数量众多，具有代表性的有以下两个。

- Jenkins[②]
- TravisCI[③]

Jenkins

原名为 Hudson[④]的 Jenkins 如今可以毫不为过地说是 CI 工具事实上的标准。全世界的项目都在使用 Jenkins（图 5.4）。

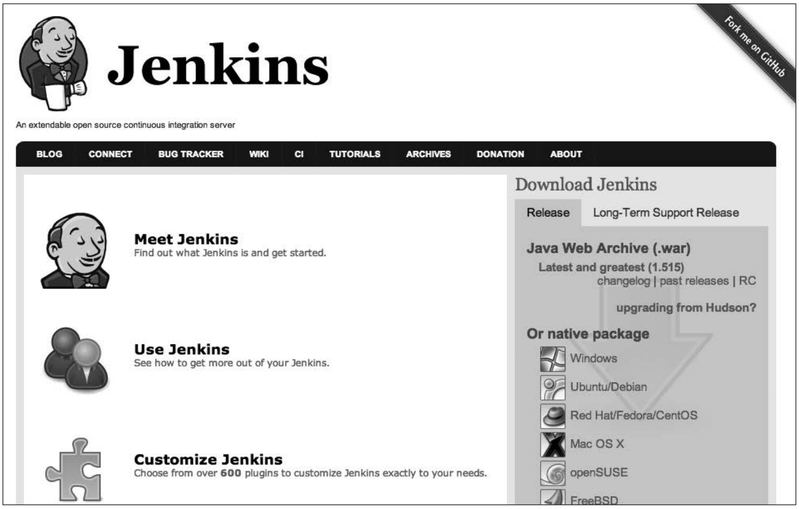
① 和 Java、Ruby 这样通用的程序不同，DSL 是为了解决特定领域的问题而设计的语言，因此也称为领域特定语言。

② http://jenkins-ci.org/

③ https://travis-ci.org/

④ 原本是 Sun Microsystems 旗下的作为 OSS 开发的名为 Hudson 的工具。由于 Oracle 收购 Sun Microsystems 而产生的商标问题，开源社区在 2010 年对 Hudson 进行了 fork（复制代码并作为另外的项目重新开始），并将其改名为 Jenkins。之后几乎所有 Hudson 的提交者都转移到了 Jenkins 项目，Oracle 也因为无法继续维护 Hudson 项目，最终于 2012 年末将其转交给了 Eclipse 基金会。后来就几乎没有向 Hudson 项目的代码提交了，而 Jenkins 则作为名副其实的活跃着的项目留了下来。

图 5.4 Jenkins

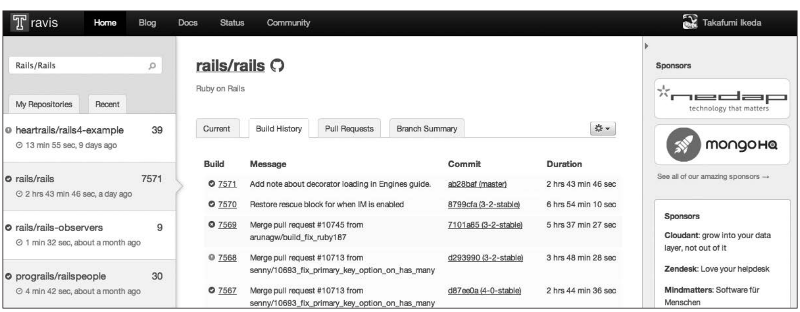


在 Jenkins 之前还有过 Continuum 和 Cruise Control 等，几乎都已经没有存在感了。从 Jenkins 被称为事实上的标准可以看出，Jenkins 具备丰富的功能并且安装简便。Jenkins 的相关内容将在 5.4 节进行讲解。

●········ TravisCI

作为通用的 CI 工具，Jenkins 可以说是一个非常不错的选择，但最近使用 TravisCI 的情况也开始逐渐增加（图 5.5）。TravisCI 是 GitHub 提供的配套的 CI 工具。虽然非 GitHub 的项目无法使用，但它设置简单，只需要 5 分钟左右就能够开始 CI。

图 5.5 TravisCI



TravisCI 的功能虽然并不多，但就进行单元测试和集成测试来说已经足够了。并且还能够和 GitHub 密切关联，使 GitHub 项目最近的 build 状态一目了然（图 5.6）。

图 5.6 用图标表示 build 状态



在合并 Pull Request 之前能够自动执行测试，这是 TravisCI 的特长[①]。借助上述功能，就能够避免在合并 Pull Request 之后才发现 build 出错的事态，使得高效的团队开发成为可能。

以前 TravisCI 只能用于公开的代码库，从 2013 年开始私有的代码库也能使用了。因此如果在使用 GitHub 的话，TravisCI 可以说是值得考虑的工具。

① 顺便提一下，Jenkins 通过使用插件也能够实现同样的功能。这部分内容将在 5.4 节进行讲解。TravisCI 默认支持 Pull Request 的 CI，这点非常方便。

5.2 build 工具的使用方法

本节将对实现 CI 所需的 build 工具的使用方法进行简单的讲解。

虽说 build 自动化是实现 CI 所必不可少的，但是没有实现 build 自动化的开发现场并不在少数。像 Java 这样比较多地使用 IDE（Integrated Development Environment，集成开发环境）进行开发的现场，特别普遍。

IDE 将编译（compile）、build、程序的启动，到测试的执行等所有流程进行了半自动化封装，用起来的确非常方便。但几乎所有的操作都要通过 GUI 来设置，在实施 build 自动化时反而会成为阻碍。与之相对，使用脚本语言的开发现场由于缺乏 IDE 的支持，比较多见的是从一开始就使用 build 工具来实施自动化。

还有一些开发现场是开发人员用 IDE 进行 build，负责 build 的人员用 Ant 这样的工具重新 build 后再实施 QA 和部署。这样的做法会造成开发时和 QA 时的 build 方法产生差异，所以应该避免这样的做法。

确保开发人员、测试人员、负责 build 的人员、负责发布的人员都能使用相同的方法进行 build，这对于 CI 的实现是非常重要的。因此开发人员也应该从一开始就使用 build 工具来构建开发环境。

build 工具种类繁多，请根据使用的语言以及开发环境来选择合适的 build 工具。

5.2.1 新建工程的情况

这里以 Java 为例进行讲解。

开始新的工程，使用 Maven 等工具从零建立工程的雏形。

Maven 是根据 CoC（Convention over Configration，惯例优先原则）的思想而设计的 build 工具。其目录的构成等都有明确的规定，能够高

效地进行 build。Maven 的安装方法请参考相关网站[①]或书籍[②]。撰写本书时（2014 年 3 月）Maven 的最新版本是 3[③]。

Maven 工程雏形的建立方式有两种：从命令行建立和通过 IDE 的 GUI 建立。

Maven 还是一款能够和 Eclipse 以及 IntelliJ IDEA[④]等 IDE 紧密结合的工具，能够通过 IDE 的 GUI 进行设置。但 Maven 原本是作为命令行工具设计的，因此应该尽可能地通过命令行来调用，这样会省去很多麻烦。

这里将简单介绍从命令行建立工程雏形到导入 Eclipse 的过程。

●……… 建立工程雏形

Maven 安装完成后，执行以下命令来建立工程雏形[⑤]。

```
$ mvn archetype:generate
```

稍许等待后会出现如下使用何种工程雏形的询问。本书执笔时共有 914 种雏形可供选择。这里使用 Maven 建议的最基本的 344 号工程雏形。确定后按下回车键。

```
略
913: remote → tk.skuro:clojure-maven-archetype (A simple Maven archetype
for Clojure)
914: remote → uk.ac.rdg.resc:edal-ncwms-based-webapp (-)
Choose a number or apply filter (format: [groupId:]artifactId, case
sensitive contains): 344:
```

接着会询问雏形的版本、groupId、artifactId 等必要的信息。

```
Choose org.apache.maven.archetypes:maven-archetype-quickstart version:
1: 1.0-alpha-1
```

---

① http://maven.apache.org/

② *Apache Maven 3 Cookbook*，Srirangan 著，Packt Publishing，2011

③ 国内的相关书籍以及网站上的内容几乎都是关于 Maven3 的，这点请注意。Maven3 最大限度地考虑了和 Maven2 之间的兼容性，因此在 Maven2 上能够运行的在 Maven3 上基本都能运行，但也有例外。还需要注意 Maven1，Maven1 和 Maven2、Maven3 并不兼容。参考网络上的资料时，请一定要注意 Maven 的版本。——译者注

④ http://www.jetbrains.com/idea/

⑤ Maven 会首先从互联网上下载必要的模块，并 build 执行命令的环境。因此第一次启动 Maven 时会执行下载和 build，需要等待一段时间。

```
2: 1.0-alpha-2
3: 1.0-alpha-3
4: 1.0-alpha-4
5: 1.0
6: 1.1
Choose a number: 6:
Define value for property 'groupId': : com.gmail.ikeike443
Define value for property 'artifactId': : sample
Define value for property 'version':  1.0-SNAPSHOT: :
Define value for property 'package':  com.gmail.ikeike443: :
```

首先要选择 maven-archetype-quickstart的版本，使用最新的版本（这里是 6:1.1），然后按下回车。

groupId 会指定区分团队或项目等多个产品的 Id。通常将所属团队的域名等用点分割并倒序显示。这里使用笔者自己的邮件地址。

artifactId 为项目自身的名字，请根据项目设置合适的名字。version 这次直接使用默认值即可。按下回车。package 通常可以和 groupId 相同，因此使用默认值，按下回车确认即可。

最后是配置内容的确认，没有问题的话请按下 Y 键。

```
Confirm properties configuration:
groupId: com.gmail.ikeike443
artifactId: sample
version: 1.0-SNAPSHOT
package: com.gmail.ikeike443
 Y: : Y
```

若输出 BUILD SUCCESS，就算成功了。Maven 会生成名为 sample 的目录，进入该目录。

```
$ cd sample
```

可以确认 sample目录下生成了如下目录结构。

```
$ tree
.
├── pom.xml
└── src
    ├── main
    │   └── java
    │       └── com
    │           └── gmail
    │               └── ikeike443
    │                   └── App.java
```

```
    └── test
        └── java
            └── com
                └── gmail
                    └── ikeike443
                        └── AppTest.java

11 directories, 3 files
```

src/main/java 是程序代码的目录。src/test/java 是测试代码的目录。像这样，Maven 能够将程序代码和测试代码存放在同一工程的不同目录下。请注意这里的“同一工程”是重点。

为了有效地执行 CI，将应用程序代码和测试代码一同置于版本管理系统中是非常重要的。Maven 就是一款能够简单地实现上述需求的工具。其他的语言也有类似的工具，可以自行选择合适的工具。

Maven 的构成如表 5.2 所示[①]。

表 5.2　Maven 的构成

目录 / 文件	说明
pom.xml	工程的 build 定义
src/main/java	程序代码
src/main/resources	程序资源
src/main/webapp	WEB-INF、JSP 文件等 Web 相关的资源
src/test/java	测试代码
src/test/resources	仅供测试使用的资源
target	Class、jar、测试结果等文件

●……… 依赖关系的定义

第 3 章中已经提到过，通过在 pom.xml 中使用 <dependencies> 标签定义依赖关系，Maven 会处理所依赖库的传递依赖，将所需要的库添加到 classpath 中。

例如本次的工程雏形中，像下面这样定义 junit 的依赖关系。

```
<dependencies>
```

① Maven 还可以用于 Java 以外的 JVM 语言的 build。这时 Maven 一般会在 Java 目录的同一层次中建立以 groovy、scala、jruby 等语言命名的目录。

```
  <dependency>
   <groupId>junit</groupId>
   <artifactId>junit</artifactId>
   <version>3.8.1</version>
   <scope>test</scope>
  </dependency>
</dependencies>
```

这里定义了 JUnit3.8.1，由于 3.8.1 太旧了，请手动改写为 4.1.1 等当时的最新版本。

那么就让我们来试着编译一下。执行如下命令。

```
$ mvn compile
```

编译后 Maven 会在 target 目录下生成 Class 文件。

●……… 执行测试

这次我们来试着执行测试。虽然 AppTest.java 依赖了 junit，但因为已经在 pom.xml 中定义了依赖关系，所以可以直接执行测试。

```
$ mvn test
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------
[INFO] Building sample 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------
略
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.gmail.ikeike443.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.049 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
[INFO] Total time: 13.282s
[INFO] Finished at: Thu May 23 18:06:32 JST 2013
[INFO] Final Memory: 15M/145M
[INFO] ------------------------------------------------------------
```

Maven 会自动处理对 JUnit 的依赖，编译代码并执行测试。

测试结果会在 target/surefire-reports/目录下生成，以文本文件和 XML 文件两种形式存储。这个 XML 文件在 Jenkins 生成报告时会起到作用。

使用 Maven 插件，测试结果报告还可以自定义为 HTML 等各类形式，这方面的内容请参考相关资料。

●········ 导入 Eclipse

最后让我们试着导入 Eclipse。运行以下命令即可生成 Eclipse 专用的 .classpath 和 .project 文件。

```
$ mvn eclipse:eclipse
```

只要从 Eclipse 的菜单导入生成的 .project 文件，就可以用 Eclipse 打开 Maven 工程了。IntelliJ IDEA 和 Netbeans[①]也有支持同样功能的插件，可以根据需要使用。

相反，通过 IDE 的 GUI 也可以初始化 Maven 的工程。Eclipse 从 3.7 版（Indigo）开始默认支持 Maven[②]。基本上通过新建工程向导就能够建立 Maven 工程。根据向导的提示，一步一步地执行即可。再重复一下，充分使用命令行对于有效地执行 CI 是非常重要的，因此尽可能地通过命令行来使用 Maven。

5.2.2 为已有工程添加自动 build 功能

新建工程时，使用 Maven 这样的 CoC 的 build 工具来初始化工程是最简单的，但同样存在已有的没有使用 build 工具的工程。这样的情况下，如果是 Java 工程，可以使用 Ant 进行 build。

Ant 不像 Maven 这样有着固定的规则。另外，因为 Ant 运行在 JVM 之上，所以有着不同于 Make 依赖于执行环境的特点。从目录的构成到依赖关系的处理、测试的执行都可以自由地进行定义。因此根据已有的

① https://netbeans.org/

② 3.6 版之前的版本需要另行安装名为 m2eclipse 的插件。

工程编写 build.xml 即可。

Ant 默认定义有 `compile`、`test`、`package`这样的任务，可以利用这些任务快速地进行 build。

但是 Ant 没有默认进行依赖关系管理，因此推荐使用 Ivy[①]这款工具。Ivy 是将 Maven 的依赖关系管理功能独立出来的工具，可以结合 Ant 使用。

5.2.3 build 工具的总结

为了实施 CI，首先实现 build 的自动化是非常重要的，可以说是必不可少的。这里以 Java 工程为例，简单地介绍了 build 工具的使用方法。

新建工程请使用 Maven，已有工程请使用 Ant，来开始实施 build 自动化。也可以使用 Gradle。不管选择哪款工具，实现 build 的自动化是非常重要的。

利用 Java 开发 Web 应用程序的情况下，通过结合 Maven 或 Ant 对 jetty、Struts2、Spring、Hibernate 这样的 servlet 容器或各类框架进行 build，可以实现 build 的自动化。

另外也可以使用全栈式的 Play Framework[②]作为 Java 的 Web 应用程序框架。Play Framework 虽然不使用 Maven 和 Ant，但默认支持自动化编译以及自动化测试的机制，因此是非常适合于 CI 的框架。

① http://ant.apache.org/ivy/

② http://www.playframework.com/

5.3 测试代码的写法

自动化测试可以说是 CI 的重点。如同我们在第 2 章问题 5 中所见，如果不持续地执行自动化测试，就无法发现不知不觉中发生的退化。

同样还有问题 7，因为缺乏测试，所以无法进行重构。通过实施 CI，构筑起能够时常自动执行测试的环境，那样就应该有信心进行重构了。从长期来看也能够有效地提高开发质量。随着质量的提高，添加新功能的速度也会相应地加快，并最终向客户提供更多的价值。

虽说 CI 的自动化测试能有效提高开发速度及质量，但为了执行自动化测试，必须编写测试代码。

请使用之前叙述过的 TDD 或 BDD 等测试框架来编写测试代码。还有一些 Web 应用程序框架原生自带测试框架。

5.3.1 作为 CI 的对象的测试的种类

一般来说软件测试可进行如下分类[①]。

- **单元测试（Unit Test，UT）**
- **集成测试（Integration Test，IT）**
- **用户验收测试（User Acceptance Test，UAT）**
- **回归测试**

其中哪些应该作为 CI 的对象呢？答案是全部。当然测试的搭建是一项耗费成本的工作，因此要根据项目的状况，仔细考虑希望得到的测试效果和成本之间的平衡。但是如果成本允许的话，还是应该将上述所有测试都作为 CI 的对象。

① 除此之外还有性能测试、β 用户测试等各类测试，这里就不提了。

本章对主要由开发人员编写的单元测试和集成测试的 CI 实施进行说明。用户验收测试的相关内容将在第 7 章进行讲解。而利用 CI 持续地执行测试可以说就是回归测试。

5.3.2 何时编写测试

何时编写测试代码？根据时间以及项目状况的不同，需要注意的事项也有所不同。

●……… 新建工程的情况

在 5.2 节中已经讲解过，新建工程时，请将测试框架和 Maven 等 build 工具一起导入。也可以使用 Ruby on Rails 或 Play Framework 等全栈式 Web 应用程序框架。每次新建类或添加方法时，都要注意将测试代码一同添加，并一同提交到版本管理系统。

编写测试代码，从短期看来，很多人担心会影响开发速度。这样的想法是不正确的。在工程代码的基础上还要编写测试代码，这的确会耗费时间，因此仅从完成编码的时间来看的确是延长了。但是编写测试代码是开发人员必须进行的重要工作，因为开发人员的工作并不是只要完成编码就结束了，而是要向顾客提供价值。换言之，就是要编写按照预期运行的软件，提供没有退化的产品。

使用测试优先或者需求优先的开发方式，就能够在编写代码的同时确认程序是否正常运行，进而使返工减少，开发的整体速度加快。

●……… 已有工程中没有测试的情况

很多已有的工程都没有实施自动化测试。根据工程规模的不同，应该采取的措施也有所差异，但还是应该尽可能地为能够测试的部分编写测试代码。

但未经测试的代码往往是无法编写或者很难编写测试的代码。类没有得到合理地职责分割，与 RDBMS 或外部系统的 API 紧密耦合，不启动整个程序就无法确认动作，这些都是常有的事情。在这样的情况下，

首先请编写从最外侧确保程序动作的测试代码。这里的“最外侧”是指用户所看到的用户界面。

Web 应用程序方面，有一款名为 Selenium 的模拟浏览器动作并进行测试的著名工具。首先可以使用 Selenium 编写针对浏览器用户界面的用户验收测试，并实施自动化。重复上述测试即等同于实施了回归测试，这样就不必再担心退化的发生，能够放心地对代码进行重构。在借助 Selenium 确保对外的用户界面不变的基础上，内部逐渐转变设计，分割为能够测试的类。这样既能逐渐提高质量，添加新功能的速度也会相应加快。

使用 Selenium 进行用户验收测试的相关内容，会在第 7 章进行详细讲解。

●……… 修改 bug 或添加新功能的情况

无论是一开始就编写了测试代码这样幸运的工程，还是没有测试代码这样悲惨的遗留工程，都会在修改 bug 或者增加新功能时添加代码。这时请一定要编写测试代码。修改 bug 的话，可以仅以修改的函数为对象添加数行测试代码。千里之行始于足下。即便只是一点一点地积累测试代码，也总有一天会覆盖全体代码。

编写测试代码还涉及工程的文化层面。也就是说，缺乏测试的工程是没有编写测试文化的工程。为了改变这样的文化，实际编写测试代码并展现其效果是最好的办法。因此在修改 bug 或添加新功能时请一定要编写测试代码。

5.3.3 棘手的测试该如何写

为了对和 RDBMS 或外部系统的 API 耦合的部分这种比较难测试的部分进行测试，这里介绍一些小技巧。

●……… 和外部系统有交互的测试

连接 RDBMS 等数据库的部分，或者调用 Twitter、Facebook 的 API

的部分等，编写依赖于应用程序外部状态的测试是比较困难的。这部分的测试要怎么写才好呢？

使用 mocking 框架进行测试

关于这部分测试，单元测试基本上是利用模拟对象 (mock) 或桩程序 (stub) 进行测试的。

例如，可以使用名为 Mockito[①] 的 mocking 框架来模拟连接数据库的部分。大致的代码如下所示[②]。

```
// 检查数据库中的用户的年龄是否满18岁
// 编写Authrization类的单元测试

// 将数据库连接对象转换为mock对象
User mocked = mock(User.class)
// 根据测试目的设置mock对象的返回值
when(mocked.getAge()).thenReturn(16);
// 在测试对象中注入mock对象并开始测试
Authorization auth = new Authorization(mocked);
// 确认年龄检查的结果
assertThat(auth.proveAge(), is(false)
```

`mock`、`when`、`thenReturn`都是 Mockito 提供的方法。这里建立 User 类的 mock 对象，在该对象的 getAge 方法被调用时，返回值定义为 16。这里不再做更详细的说明，上述测试代码的写法还是相当直观、易懂的。Mockito 是一款功能强大的 mock 框架，详细的使用方法请参考相关资料。

这样，本来不得不根据测试用例来修改数据库值的地方，借助 mock 框架便不必实际修改数据库，使用 mock 对象就能够编写测试。

虽然没有列举示例代码，但和 Web 服务的 API 相关的部分同样可以使用 Mockito 来模拟。例如，可以为连接 Web 服务的类制作 mock，使其返回固定的 JSON 作为响应。

CI 中，由于同一测试要持续地、反反复复地执行，因此无论执行几

---

① http://mockito.org/

② 这里的示例代码非常简单。在实际工程中编写这样的 mock 的测试代码时，对象类中获取外部数据的部分需要能够简单地替换成 mock，需要具备 DI（Dependency Injection）的机制。DI 的形式多种多样，这方面可以参考相关资料。

次都必须返回相同的结果。数据库是将状态持久化的工具，所以反复执行测试的话，数据库的状态会发生变化。防止这一问题的方法之一就是 mock。如前所述，mock 能够提高单元测试的效率，不仅是数据库关联的测试，调用外部 Web 服务 API 的测试同样可以使用。

如果不使用 mock 框架，也可以在每次进行测试时启动数据库、建立数据库、CREATE 表格、加载数据，测试结束后再将数据库删除。已经存在大量代码没有测试的情况下，添加 mock 的机制比较困难，因此多采用上述方法。

例如，在使用 Selenium 实施用户验收测试时，比起使用 mock，使用真正的数据库进行测试或许更有价值。用户验收测试中数据库连接是否成功本身也是需要测试的项目之一，多数情况下，表之间复杂的状态迁移也需要进行测试。并且在进行用户验收测试时，如果要将所有的相关对象都 mock 化，那么所涉及的数据条数以及表的数量实在太大，实际上很难做到。

这样的情况下只能采取在每次测试时构建并删除数据库的方法。由于这样的处理非常费时，测试的速度也会由此变慢。并且在真实的数据库上建立这样的机制要耗费不少的时间和精力[①]。

●……… 使用内存数据库进行测试

这里有必要考虑使用一款名为 H2 的数据库。它是用 Java 编写的轻量级的数据库引擎，支持内存数据库模式。使用 H2 能够使得重量级的数据库构筑处理变得轻便，测试的范围也得到了相应的拓展。需要注意的是，H2 无法支持各个 RDBMS 的独特的检索语法，关于这一点，可以在中间加一层 Hibernate 这样的 O/R 映射工具，来吸收各个数据库之间的差异。

现代的 Web 应用程序框架应该能够根据运行时的配置更改数据库的连接。例如 Play Framework 在执行测试时默认使用 H2 连接数据库[②]，并

① 这样的情况下可以使用 DbUnit 等工具，虽然这些工具有些旧了，但还是多少能够节省一些时间。

② 当然配置可以修改。

且还提供了名为 Fixture 的数据加载机制。借助该机制，执行测试时就可以用和正式环境相同的步骤，在内存上高速地构建数据库和表，加载数据，在此基础上，编写的测试用例也能够以和正式环境相同的方式来访问数据。因为是内存数据库，所以能够在测试结束后轻易地销毁。测试也因此具备了很高的独立性。写过测试代码的各位一定知道，和数据库紧密耦合的话，每次测试后数据库状态都会发生变化，因此测试非常难写。而内存数据库将很大程度地改善这个问题。

●……… 数据库变更管理和配置文件管理的测试

如第 3 章中所提到的那样，在利用数据库迁移机制进行数据库的变更管理的情况下，SQL 的正确与否可以通过表的生成是否正确来判断，但应用程序的动作是否正确还不得而知。

数据库迁移自身的测试也可以简单地写一下。数据库构筑完成后，写一下简单的冒烟测试[①]，只要能够确认应用程序的基本部分运行正常就可以了。

环境构建部分的测试同样也是写一下比较好。中间件启动后，确认一下是否有响应，即使是这样简单的测试，也是非常有效的。关于使用 Chef 或 serverspec 的环境构建自动化以及相关的测试详情，我们将在第 6 章讲述。

数据库的变更管理也好，环境配置管理也好，为测试这些项目，严密地说需要在这些项目的基础上来确认应用程序的所有动作。从这个意义上来说，如果要对这方面进行严密的测试的话，通过用户验收测试来确认是比较正确的做法。当然也要根据工程的状况、预算以及期限适度地编写测试。

●……… UI 相关的测试

UI 相关的测试是比较困难的部分。因此要尽可能地将业务逻辑内聚在 MVC 模式中的 M（模型）中，设计为不必涉及 UI 就能网罗几乎所

---

① 简单的动作确认测试。为了确认是否能够实施正式的测试而进行的准备性质的测试。原本是电机行业的术语，指将冰箱、电视机等接通电源看看有没有冒烟。

有的测试用例。应该尽量避免编写依赖于 UI 的测试。

最近使用充分挖掘了 Ajax 特性的 Rich UI 的 Web 应用程序也开始多了起来。随之而来的是，JavaScript 的 MVC 框架也开始大量出现。如果在客户端较多地使用了 JavaScript 的话，应该尽量将业务逻辑内聚在 M 之中，以便可以通过 UI 以外的部分进行测试。

不得不针对 UI 编写测试的情况下，可以利用 Selenium 这样的工具来实现用户验收测试的自动化。使用 Selenium 的用户验收测试自动化的相关内容将在第 7 章详细讲解。

●……… 棘手的测试要权衡工数

编写和外部交互相关的测试或数据库变更管理、UI 相关的测试等相当棘手的测试自然会耗费相当多的工数。如果只考虑质量的话，这些测试的确是完整地写一下比较好，但实际上编写测试可用的时间并不是无限的。

测试的自动化是越做越耗费时间和精力。将编写测试所获得的质量提升以及能够在未来消减的工数，与编写测试实际耗费的工数进行权衡，注意保持两者的平衡。

如果过度执着于测试自动化的工作，其本身就像填字游戏一样，虽然有趣，但不知不觉之中就可能做过了头，投入和回报不相符了。

请大家根据自身所处的状况，以及项目所追求的价值，在最合适的限度内来实现测试的自动化。

5.4 执行基于 Jenkins 的 CI

已经集齐了所需要的版本管理系统、自动 build 以及测试代码之后，接下来就让我们实际使用 Jenkins 来实施 CI 吧。

5.4.1　Jenkins 的安装

Jenkins 的安装非常简单。从主页[①]下载 WAR 文件并执行以下命令即可。

```
$ java -jar jenkins.war
```

启动 Jenkins 后访问 `http://localhost:8080`，若显示如图 5.7 这样的画面，即说明安装成功了。

图 5.7　Jenkins 安装完成画面



① http://jenkins-ci.org/

●……… 使用本地安装包进行安装

在刚才的例子中，为了体现启动 Jenkins 有多么简单，选择了直接使用 WAR 包启动。而在实际的开发现场，则是将 Jenkins 作为 OS 的服务（UNIX/Linux 环境的守护程序）启动比较好。Jenkins 为各个 OS 提供了安装包。例如，使用 Windows 平台的安装包进行安装就能作为 Windows 的服务启动，使用 CentOS 的安装包就能作为 Linux 的驻守程序启动。Jenkins 的主页上记载有各个 OS 的本地安装包的安装方法，可以参考。

这里以 Red Hat Enterprise Linux 为例进行讲解。新建 Jenkins 用的目录并执行 yum命令即可。启动脚本也会一并安装。

```
$ wget -O /etc/yum.repos.d/jenkins.repo http://pkg.jenkins-ci.org/redhat/
jenkins.repo
$ rpm --import http://pkg.jenkins-ci.org/redhat/jenkins-ci.org.key
$ yum install jenkins
```

执行如下命令，即可将 Jenkins 作为 OS 的守护程序启动。

```
$ /etc/rc.d/init.d/jenkins start
```

和使用 WAR 启动时一样，启动后请试着访问 `http://localhost:8080`，若显示如图 5.7 这样的画面，即说明安装成功。

顺便提一下，Jenkins 还能在 Tomcat[①]等 Servlet 容器上启动。这时会监听 ajp 的 8009 端口，所以用 proxy_ajp 等作为代理上传至服务器即可。

5.4.2　Jenkins 能干些什么

Jenkins 是一款为实施 CI 提供支持的工具，通过和各类插件组合使用，可以实现非常丰富的功能。详细内容可以参考 Jenkins 相关的书籍[②]。

这里对 Jenkins 所支持的最基本的 CI 进行说明。先前列举了 CI 所必需的版本管理系统、build 工具、测试代码，将这些组合起来构建如下这样的机制。

① http://tomcat.apache.org/

② John Ferguson Smart 著，Jenkins: The Definitive Guide，O’Reilly Media, Inc, USA，2011

- 下载（checkout）代码
- 自动 build 并执行测试
- 统计结果并制作报表
- 通知

让我们试着来建立一种机制，实现当代码有更新时能够立即并且持续地执行上述一系列处理。

5.4.3 新建任务

Jenkins 以任务（Job）为单位来管理一系列的处理流程。可以姑且理解为 1 个应用对应 1 个任务，或者 1 个目录对应 1 个任务[①]。

首先在 Jenkins 上新建任务。点击菜单上的“创建一个新任务”，会出现“构建一个自由风格的软件项目”“构建一个 Maven2/3 项目”“构建一个多配置项目”“监控一个外部的任务”“复制已有的 Item”5 个选项，这些选项各自的内容在 Jenkins 的主页上都有说明，请自行参考。

如果使用 Maven bulid 工程的话，可以选择“构建一个 Maven2/3 项目”。Jenkins 支持向 Maven 目录公开（publish）所生成的文件之前的流程。而如果是 Ruby、Node.js 等 Java 之外的项目，或者是使用 Maven 以外的 Java 项目，请选择“构建一个自由风格的软件项目”。

5.4.4 下载代码

下面进行从版本管理系统下载（checkout）代码的配置。虽然这里写的是“代码”，但是应该进行版本管理的并不仅限于代码，还包括数据库模式的定义和配置文件等应用程序启动所需的所有资源。用 Jenkins 将这些资源下载到干净的环境。

上述内容在代码管理这一项目中进行配置。Jenkins 默认可选的代码管理系统为 CVS 和 Subversion，使用其他的版本管理系统需要安装插件。

这里以使用 Git 作为前提进行讲解。点击菜单中的“系统管理”，打

① 实际上 1 个任务也可以操作多个目录，通过配置可以实现各种不同的运用。

开“管理插件”，在“可选插件”标签下查找名为“Git Plugin”的项目（图 5.8）。

图 5.8 Git Plugin



点击“下载待重启后安装”按钮，重启 Jenkins，完成安装。再次打开任务的配置页面，确认一下代码管理的地方，可以看到增加了名为 Git 的选项。按照图 5.9 进行配置。

图 5.9 Git 的配置

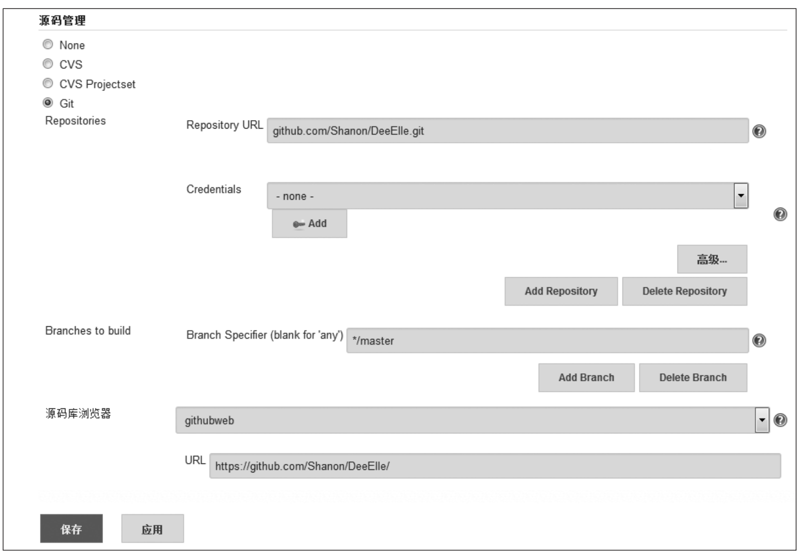


图 5.9 中，在“Repository URL”中填写 Git 的 URL，在“Branches to build”中填写 build 对象的分支名称。然后在“源码浏览器”处填写阅览 Git 时使用的工具，这里选择了 githubweb，其他还有 Gitlab 和 Gitrious 等选项。

5.4.5 自动执行 build 和测试

下面对 build 进行配置。分别对“构建触发器”和“构建（build）”进行设置。

在“构建触发器”中配置执行 build 的触发器。

●……… 定期执行

比如 1 天执行 1 次、每隔 4 小时执行 1 次等，设置为定期自动执行 build 的方式。可以用近似于 cron 的写法来设置执行的频率。

定期执行是从还没有 CI 这个词汇时就有的手法。例如 *Joel on Software*[①]中写到了 Microsoft 从 1990 年代开始 1 天进行 1 次 build（daily build）。还有描写 1990 年代初 Windows NT 的开发的《观止——微软创建 NT 和未来的夺命狂奔》[②]，其中也提到了 daily build。

daily build 严格来说并不是 CI，但配置简单，实施起来也算是不错的方式。1 天之内提交的量并不是太多的话，进行 daily build 也不会有什么问题。也有一些案例是因为 build 或测试的执行时间过长，导致无法及时 build 而不得不采用 daily build 的方法。定期执行作为 CI 来说虽然有所欠缺，但比起什么都不做来说还是要好很多的。

●……… 轮询版本管理系统

即配置为以轮询（polling）的方式监视版本管理系统，有新的提交（Push）时就执行 build 的方式。能够用类似于 cron 的写法来指定轮询的

① Avram Joel Spolsky，*Joel on Software*，APress，2004 年

② 《观止——微软创建 NT 和未来的夺命狂奔》G. Pascal Zachary 著，张银奎译，机械工业出版社，2009

频率。由于在每次提交时都会进行 build 及测试，因此能够更早地发现问题，有助于质量的提高。

因为采取的是轮询的方式，所以无论是否有提交都会向版本管理系统发送请求。虽然这会给系统造成不必要的负担，但只要没有太大的问题，采用轮询的方式还是能够比较方便地实施 CI 的。

build 执行的频率增加后，build 的速度就变得很重要。过于缓慢的话，提交集中时，build 会积累在任务队列中，变得难以及时地察觉问题。为了避免这样的情况，就需要在 build 或测试代码上下功夫，提高 CI 服务器的处理速度，增加服务器台数实行并行 build 等。

尽可能地避免定期执行，以每次提交时进行 build 这样的循环来实施 CI 是很重要的，因为这样版本管理系统的提交记录和 CI 服务器的 build 结果就能关联起来，工程的可追溯性能够得到提高。

专栏　从版本管理系统进行 Push

从 Jenkins 的配置来说，“轮询版本管理系统”是最适合于 CI 的，但最好能够由版本管理系统通过 Push 来请求 Jenkins 执行 build。Git 和 Subversion 都能够配置 Push，这里介绍下使用 Git 的方法。

Git 提供了在发生提交或收到 Push 等事件时执行特定脚本这样的钩子（hook）的机制。在各个代码库下的 .git/hooks/ 目录下有示例文件，编写自己的脚本时可以进行参考。详细内容请见 *Pro Git*[①]。Subversion 也有钩子的机制，可以实现相同的功能。

例如，在代码被 Push 到 Git 代码库时，若要执行 Jenkins 的任务，可以建立 .git/hooks/post-receive 文件，并按照下面的内容进行编辑。上述处理是以安装 Jenkins 的 Git Plugin 为前提的。

```
#!/bin/sh
curl http://{Jenkins 服务器的域名或 IP 地址等 }/jenkins/
git/notifyCommit?url={Git 代码库的 URL}&branches={ 分支名（可以设置
多个）}
```

curl 部分也可以是 wget 等命令，只要配置后能够向 Jenkins 发

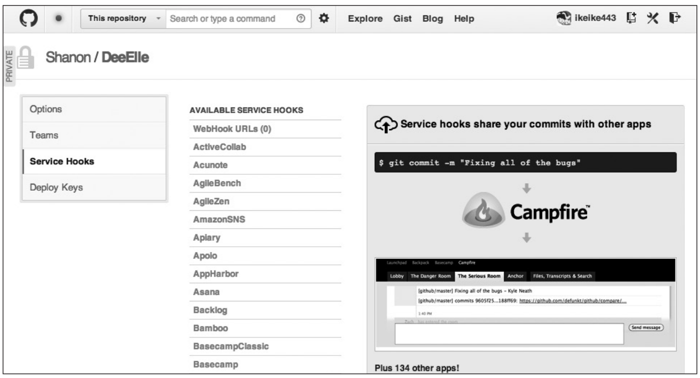
① http://git.oschina.net/progit/

送上述这样的 GET 请求即可。

通过上述配置，在 Git 代码库收到 Push 时就会向 Jenkins 发送上述 GET 请求。收到请求的 Jenkins 会执行包含对应代码库的所有任务。关于这个功能的详细内容请参考 Git Plugin 的帮助[①]。

如果使用的是 GitHub 的话，可以利用 GitHub 的 Service Hooks 设置。打开代码库 Settings 下的 Service Hooks 就能看到，GitHub 能够和全世界的各类服务进行交互（图 5.a）。

图 5.a　Service Hooks



从中选择 Jenkins，并配置好 Jenkins 的 URL，就能同样在 GitHub 收到 Push 时向 Jenkins 发送执行任务的请求。但是需要注意的是要将 Jenkins 搭建在 GitHub 可见之处。通常 Jenkins 多被搭建在公司内部的局域网（LAN）中，所以需要在网络的配置上下一些功夫。顺便提一下，GitHub 发送请求的服务器 IP 地址是公开的，只要对 Jenkins 加上 IP 限制，就能够消除安全方面的隐患。

这样的方法能够消除延迟，实现在每次提交后立即实施 build。

build 的记述

下面来讲一下 build 以及测试的处理。可以在“增加构建步骤”中选择 Ant、Maven、shell 脚本、Windows 批处理文件中的任意一项。Java 的项目可以选择 Ant 或 Maven。

① https://wiki.jenkins-ci.org/display/JENKINS/Git+plugin

如果是 Java 以外的语言的话，可以用 shell 脚本来记述 build 的处理。使用 Make、Rake 或 SBT 等 build 工具的话也没有问题。虽然在 Jenkins 的配置页面上能够直接编辑 shell 脚本，但还是建议将 build 脚本记录在文件中并提交到版本管理系统，在 Jenkins 中只做简单的调用。这样即便没有 Jenkins，也能够以完全相同的方法来执行 build，非常方便。

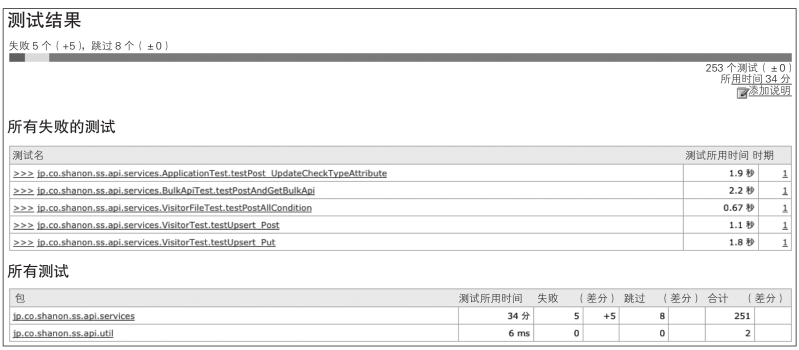
另外，如果不使用 build 工具，而是利用 shell 脚本来描述 build 的话，请设置相适应的退出值（exit code）[①]。Jenkins 根据该值来判断任务的执行是失败还是成功。

5.4.6　统计结果并生成报表

可以在“构建后操作”中选择结果的通知方式。

可选的项目各种各样，这里为了便于统计测试结果，请选择“Publish JUnit test result report”。随后就会出现设置测试结果 XML 文件的路径的输入框，输入测试结果 XML 文件的路径即可。这样配置后再进行 build，就能如图 5.10 这样在浏览器上确认结果。

图 5.10　测试结果



测试结果

失败 5 个（+5），跳过 8 个（±0）

253 个测试（±0）
所用时间 34 分
添加说明

所有失败的测试

测试名	测试所用时间	时期
>>> jp.co.shanon.ss.api.services.ApplicationTest.testPost_UpdateCheckTypeAttribute	1.9 秒	1
>>> jp.co.shanon.ss.api.services.BulkApiTest.testPostAndGetBulkApi	2.2 秒	1
>>> jp.co.shanon.ss.api.services.VisitorFileTest.testPostAllCondition	0.67 秒	1
>>> jp.co.shanon.ss.api.services.VisitorTest.testUpsert_Post	1.1 秒	1
>>> jp.co.shanon.ss.api.services.VisitorTest.testUpsert_Put	1.8 秒	1

所有测试

包	测试所用时间	失败	（差分）	跳过	（差分）	合计	（差分）
jp.co.shanon.ss.api.services	34 分	5	+5	8		251	
jp.co.shanon.ss.api.util	6 ms	0		0		2	

从图 5.10 中可以看出，253 个测试中有 5 个测试失败了。测试所消耗的时间也能从图上看出。从“时期”这一列可以知道这是从在此之前的第几次 build 开始出现失败的，图 5.10 中的 5 个测试都是在这次 build

① 成功的话返回“exit 0”，失败的话返回“exit 9”。

中才失败的。其他还有一些针对包的测试，可以看出和上一次 build 相比较，失败的个数有所增加。

专栏　以 JUnitXML 的形式输出报表比较高效

通过使用插件，还可以对 JUnitXML 之外的测试结果进行统计。但考虑到通用性和报表的直观性，以 JUnitXML 形式输出的报表是最有效的。如果使用的测试框架是 JUnit 的话自然没有问题。如果使用的是其他测试框架，因为报表生成部分的实现大多能以插件的形式进行替换，所以这里推荐修改为以 JUnitXML 的形式来生成报表。

例如 Scala 的 Specs2 就能够通过修改配置来改变生成报表的形式。JavaScript 的 Jasmine 也有志愿者制作的名为 JUnitXmlReporter 的输出 JUnitXML 的对象[①]。其他的语言也基本都能够输出 JUnitXML，因此建议先调查清楚再着手应对。

5.4.7　统计覆盖率

至此，我们实现了到统计持续 build 及测试结果为止的处理流程。通过自动执行测试，对于测试覆盖的部分就能够确认测试是否成功，但测试没有覆盖的部分就没有任何信息。这样的情况下，让我们来统计一下代码覆盖率（code coverage）。代码覆盖率是表示测试对象的应用程序代码在测试中被执行了多少的指标。统计覆盖率能够看出代码被测试的程度，或者反过来说，能够使还需要增加多少测试这样的信息可视化。

但需要注意的是，代码覆盖率只是反映了测试对象的应用程序代码是否被执行，并不能反映出用作测试的测试代码是否妥当，因此并不可以盲目相信。但在认识到需要通过代码审阅（source review）等方式来确保测试代码的内容的正确性之后，代码覆盖率作为质量指标之一会变得很重要，所以请一定要试着用一下。

① https://github.com/larrymyers/jasmine-reporters

●……… 覆盖率统计工具

统计覆盖率需要安装专门的工具。具有代表性的覆盖率统计工具有下面这些。

- Cobertura①
- Jacoco②
- Scct③
- SimpleCov④
- Rcov⑤

这些工具是以 Ant、Maven、SBT、Rake 等 build 工具的插件的形式提供的。可以根据开发环境选择合适的工具。这次以作为 Maven 的插件提供的 Cobertura 为例进行讲解⑥。

●……… Maven Cobertura 插件的安装

在 Maven 中使用 Cobertura，只需要在 pom.xml 中定义如下依赖关系即可。

```
<dependencies>
  <dependency>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>cobertura-maven-plugin</artifactId>
    <version>2.5.2</version>
  </dependency>
</dependencies>
```

依赖关系定义完后，执行下列命令。

```
$ mvn cobertura:cobertura
```

① http://cobertura.github.io/cobertura/
② http://www.eclemma.org/jacoco/
③ https://github.com/sqality/scct
④ https://www.ruby-toolbox.com/projects/simplecov
⑤ https://github.com/relevance/rcov
⑥ Cobertura 已经于 2011 停止了开发，所以今后改用 Jacoco 比较好。因为习惯了用 Cobertura 编写报表的方式，笔者个人比较喜欢使用 Cobertura，而且现在还在使用。Java 之外的语言的覆盖率统计工具也大都支持 Cobertura 形式的 XML。

这样就会执行测试，并在测试之后对覆盖率进行统计。结果文件以 html 的形式生成在 `target/site/cobertura/` 目录下。可以在浏览器上打开 index.html 来确认报表（图 5.11）。

图 5.11　覆盖率的报表

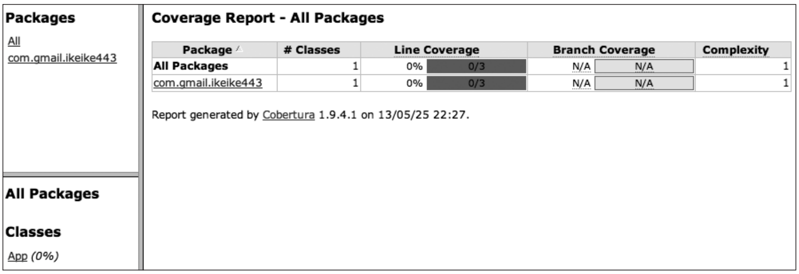


图 5.11 中因为还没有编写测试，所以覆盖率为 0%。添加测试后就能通过报表确认覆盖率上升。作为供人阅览的报表来说，这样的 HTML 形式确实看起来比较方便，但就利用 Jenkins 来实现 CI 来看，还是以计算机易于理解的 XML 形式输出文件比较方便。Cobertura 的报表的默认形式是 HTML，可以通过修改配置同时生成 XML 形式的报表。

如下这样编辑 pom.xml。

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
apache.org/xsd/maven-4.0.0.xsd">
略
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  <build>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>cobertura-maven-plugin</artifactId>
        <configuration>
          <formats>
            <format>xml</format>
            <format>html</format>
          </formats>
        </configuration>
      </plugin>
```

```
    </plugins>
  </build>
  <dependencies>
略
  </dependencies>
</project>
```

在这样的状态下再次执行 mvn　cobertura：cobertura，就会发现除了之前的 html 文件之外，还生成了名为 coverage.xml 的文件。

这样覆盖率的统计就完成了。接着让我们对 Jenkins 进行配置。

专栏　Java 程序库的查找方法

笔者参与过数个开发现场，其间所体会到的事情之一就是使用 Java 的开发现场中程序库的查找方式以及安装方法存在不恰当的地方。

在 Web 上检索相关信息，下载可能符合条件的 jar 文件，并将其加入 class path 使用，这样的事例意外的多。如此一来程序库的可信度就有问题了。

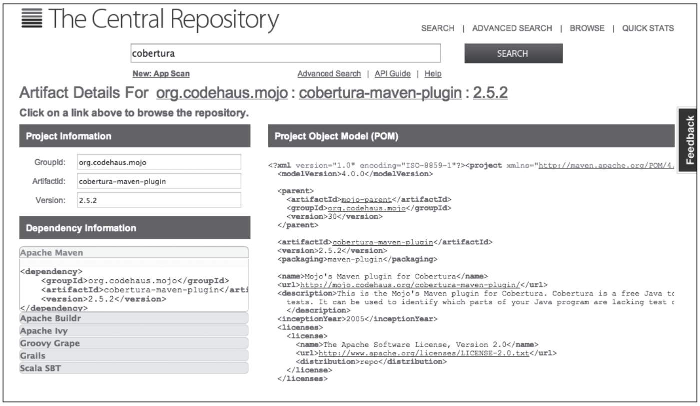
Perl 和 Ruby 这样的脚本语言，各个社区之间彻底地应用了包（package）管理。但 Java 不全是这样。脚本语言从早期就采取了包管理的机制，和 OSS 亲和性比较高。

Java 也有包管理机制，如 Maven 仓库。访问 http://search.maven.org 就可以看到很多 Java 系语言（也包括 Scala、Groovy、JRuby 等）的 OSS 程序库。

例如，搜索 Cobertura，可以找到几个符合条件的程序库。这里试着使用一下 Cobertura 的 Maven 插件（图 5.b）。

从图 5.b 可以看出如何在 Maven 的 pom.xml 中添加依赖关系来安装 Cobertura。图中还记载了使用 Ivy 和 SBT 以及其他 build 工具时的写法，请务必参考。

图 5.b Cobertura 的 Maven 插件

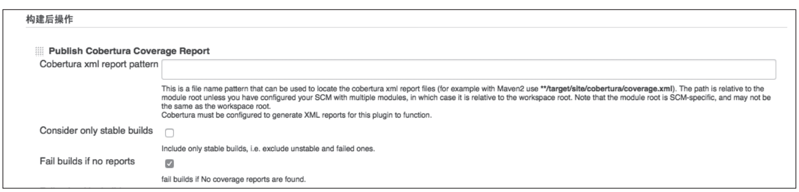


●········ Jenkins 插件的配置

能够在 Maven 中执行 Cobertura 后，接着对 Jenkins 进行配置。Jenkins 中提供了 Cobertura 的插件，和安装 Git 的插件一样，请从插件管理进行安装。

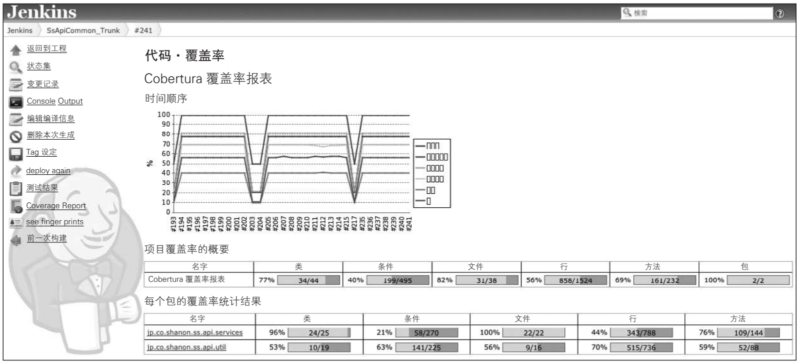
安装完成后，在任务配置的构建后操作中会增加选项“Publish Cobertura Coverage Report”，选中该项。在出现的配置栏（图 5.12）中输入 `**/target/site/cobertura/coverage.xml`，其他的配置保留默认值即可。

图 5.12 Cobertura 覆盖率报表统计



这样 Cobertura 的配置就完成了。Jenkins 会将覆盖率的报表以美观、易懂的形式呈现出来（图 5.13）。

图 5.13 Jenkins 的覆盖率报表



Jenkins 下能够看到每种度量方式的覆盖率变化的图表，以及更进一步的每个 class 文件的覆盖率。开始统计覆盖率后，开发人员编写测试代码的积极性也会有所提高。看着每次添加测试覆盖率都会有所提升，这是一件很有成就感的事情。在提高团队开发的质量的过程中，维持开发人员编写测试代码的积极性是相当重要的要素，从这一意义上来说对覆盖率的统计也是相当重要的。

5.4.8 静态分析

在自动化测试和统计代码覆盖率的基础上，若能对代码实施静态分析，就可以进一步提高团队开发的质量。因此应通过 Jenkins 持续地检查代码是否符合编码规则、是否容易产生 bug 等。

能够在 Jenkins 中使用的静态分析工具有以下这些。

- Checkstyle[①]
- PMD[②]
- Findbugs[③]

① http://checkstyle.sourceforge.net/
② http://pmd.sourceforge.net/
③ http://findbugs.sourceforge.net/

各个工具都有自己的特点。Checkstyle 擅长编码规则的检查等；PMD 的可定制化的程度高，对编码规则以及潜在的 bug 都能够检查；Findbugs 则以善于检查潜在的 bug 为特点。各个工具都支持规则的定制和扩展。请根据项目的具体情况选择合适的工具。

使用 Maven 作为 build 工具的话，静态分析工具和 JUnit、Cobertura 一样，安装 Maven 的插件即可使用。当然在 Ant 等上面也可以使用。Jenkins 的话同样只要安装插件就能使用。

通过 CI 实施静态分析的优点有很多，其中最显著的是可以减轻代码 review 时的负担这一点。代码 review 虽说是为了提高质量而需要实施的实践项目之一，但其实施的成本之高是无法忽视的。而如果能确保静态分析自动化，review 时就可以忽略对编码规则等细节的检查，从而对更为本质的内容进行 review。

面向 Java 这样的静态类型语言的静态分析工具还是比较多的，但面向动态类型语言的工具就比较少了。特别是要通过 Jenkins 来统计的话就更难了。即便如此，还是有面向 JavaScript 的 JSLint[①]等几款可以通过 Jenkins 进行统计的工具，可以根据需求试着找一下。如果没有的话，自行制作也不失为一个不错的方法。

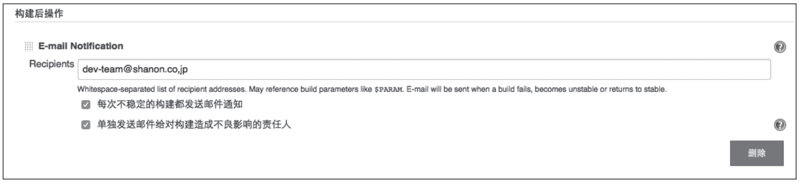
5.4.9　配置通知

最后对 build 结果的通知进行配置。包括插件在内，Jenkins 提供了多种通知 build 结果的手段。可以通过邮件、IRC、Twitter 来通知，还可以利用称为 XFD（eXtreme Feedback Device）的硬件进行通知。特别是 XFD，经过精心的设计，XFD 在 build 失败时会鸣响警报或发射玩具火箭等，非常有趣。

这里讲解一下通过邮件来通知的方式。Jenkins 默认提供了邮件通知的功能，只需要在“构建后操作”中选择“E-mail Notification”即可（图 5.14）。

① http://www.jslint.com/

图 5.14 邮件通知的设置



在收件人（Recipients）处填写邮箱地址后，在 build 失败或者失败的 build 被修复时就会收到邮件通知。可以在此配置团队的邮件列表等。默认设置中选中了“每次不稳定的构建都发送邮件通知”，这样一来，在连续 build 失败的情况下，每次都会收到通知邮件。通常这样设置是没有问题的，但如果项目经常 build 失败、状态不好，解除这个选项就不会每次失败时都收到邮件了。但这里并不建议解除该选项。

如果选中“单独发送邮件给对构建造成不良影响的责任人”，那么除了刚才在收件人处配置的邮箱地址之外，还会给造成 build 失败的当事人发送邮件。使用该功能需要预先为 Jenkins 配置邮件服务器。

5.5 CI 的运用

Jenkins 的配置完成后就可以实施 CI 了。这里我们来说一下 CI 运用上的小窍门以及同版本管理系统和缺陷管理系统的协作。

5.5.1 build 失败了该怎么办

开始实施 CI 后会由于某人的提交而造成 build 出错。这里所说的 build 出错是指代码无法编译（compile）、静态分析出现错误、测试失败等现象。

那么 build 失败的话应该怎么办呢？造成 build 失败的当事人应该对提交进行修改，避免 build 再度失败。而其他的团队成员应该做些什么呢？这个根据版本管理系统运用方式的不同而有所差异。

●……… Subversion 等中央集权型版本管理系统的情况

build 失败的话，从那一刻起就不得不禁止在相同代码库的相同分支上进行开发的所有成员的提交。对于全员都在同一代码库上进行开发的中央集权型版本管理系统来说，在造成失败的提交上再叠加其他人的提交的话，修改会变得更为困难。最糟糕的情况是在造成 build 失败的提交之上又叠加了其他有问题的提交，这样的事情也是存在的。

●……… Git 等分布式管理系统的情况

Git 的情况下原则上和 Subversion 一样。从 build 失败的那一刻起就要禁止在相同代码库的相同分支上进行开发的所有成员的提交，这是比较简便的运用方法。采用第 3 章中介绍的 git-flow 这样的工作流程，全员向同一个代码库进行 Push 的话，就需要禁止提交。

但是像 github-flow 这样，各个成员拥有自己的代码库，通过发送

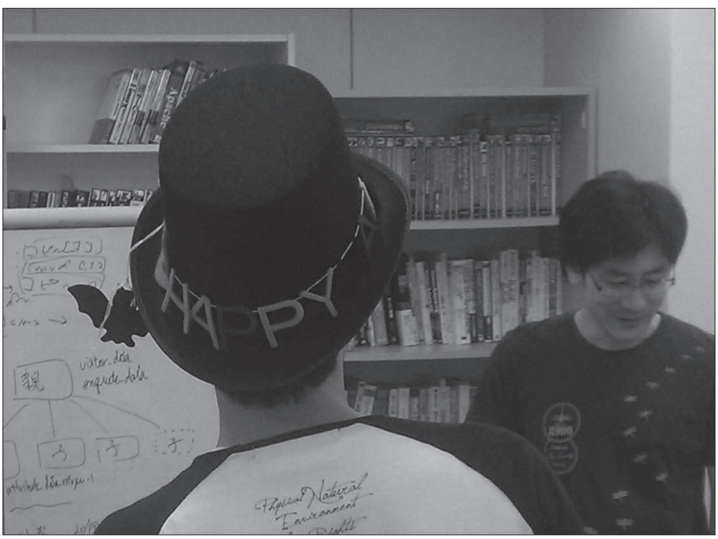
Pull Request 进行修改的话，就没有必要禁止全员提交了。只需要在 Pull Request 被发送到中央代码库时确认该 Pull Request 和中央代码库合并后的代码是否能够通过 build 即可。这也是比较可行的方法。

专栏　造成 build 失败后的惩罚游戏

笔者所在的公司中，build 失败时会向全体成员发送邮件，并且规定从收到邮件后到 build 恢复正常为止禁止全员的提交。

为了让大家知道是谁造成了 build 失败，造成 build 失败的人要戴一顶大的礼服帽作为惩罚游戏。直到 build 修复为止，要一直戴着这顶帽子，公司内部会议也只能戴着帽子参加（图 5.c）。因为这样非常丢脸，所以大家在提交时会加倍小心。但如果做过了头就变得像追查犯人一样，导致开发人员畏首畏尾，不愿意从事具有挑战性的新功能开发。因此要保持在开玩笑[①]的程度，让团队开发顺利地推进。

图 5.c　戴大礼服帽的惩罚游戏



① 开玩笑是团队开发中的重要元素。它能够促进团队成员之间的交流，使项目顺利进展。可以采用之前提过的 XFD，或者使用 Jenkins Emotional Plugin（https://wiki.jenkins-ci.org/display/JENKINS/Emotional+Jenkins+Plugin）这款会根据 build 结果改变 Jenkins 先生的表情的插件，都是非常有趣的。

●……… 测试后合并

本章在介绍 TravisCI 时介绍过，结合 GitHub 和 TravisCI 能够实现测试通过后再提交。其实 Jenkins 借助 GitHub pull request builder plugin 插件也可以实现同样的功能。这款插件可以实现的功能和 TravisCI 基本相同。插件的配置方法可能稍许有些复杂，具体请参考插件的帮助网站（英文）[①]。这里只做简单的讲解。

❶ 安装插件

采用和 Git Plugin 相同的安装方法[②]安装 GitHub pull request builder plugin 插件即可。

❷ 在 GitHub 上新建 bot 用户

插件自身为了在 Pull Request 中添加各类评论，需要 bot 用户。在 GitHub 上以适当的名字新建用户（可以使用自己喜欢的名字）。

❸ 配置 Jenkins 的系统

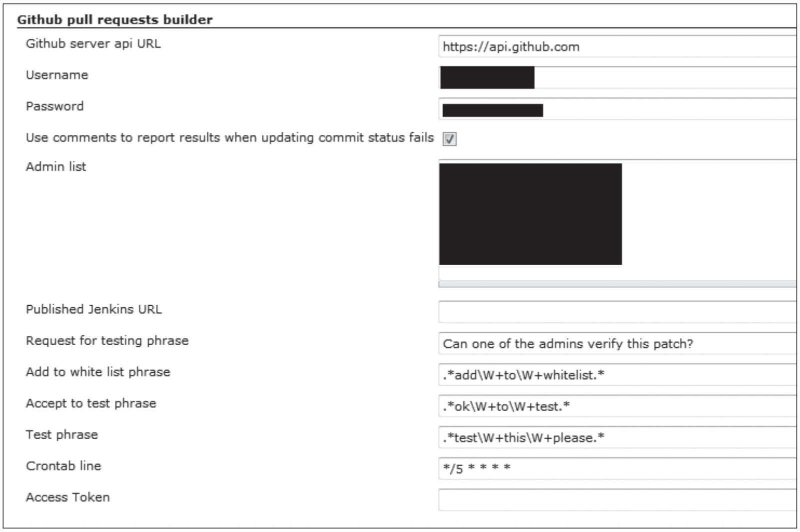
输入 GitHub API 的 URL 以及 GitHub 的用户名 / 密码（图 5.15）。除了用户名 / 密码之外，也可以输入 GitHub 的 Access Token（使用 Access Token 更为安全）。在 Admin list 中填写作为该插件的 Admin 的 GitHub 用户的列表。Admin 可以在 Pull Request 的评论上通过和 bot 用户的对话进行各类操作。其他的项目保留默认值即可。

---

① https://wiki.jenkins-ci.org/display/JENKINS/GitHub+pull+request+builder+plugin

② 请参考 5.4 节。

图 5.15　GitHub pull request builder plugin 的配置



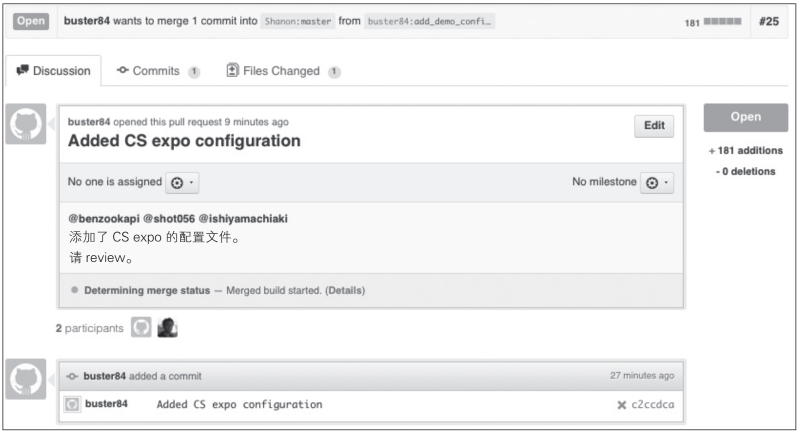
❹ 配置各个任务

对各个任务进行配置。首先，在 GitHub project 中填写对象 GitHub 代码库的 URL。代码管理系统的配置选择 Git，设置为 Repositories 的 Repository URL。“高级”的 refspec 栏填入 `+refs/pull/*:refs/remotes/origin/pr/*`。Branches to build 的 Branch Specifier 设置为 `${sha1}`。接着，在“构建触发器”中的 GitHub pull requests builder 这个选项上打钩。在 Admin list 中添加这个任务中作为 Admin 的用户信息。如果事先知道发送 Pull Request 的用户的话，可以在“高级”的 White list（白名单）中输入该用户的名称，这样配置能够简便很多。

❺ 实际发送 Pull Request

试着发送 Pull Request。对于收到的 Pull Request，中央代码库会每隔 5 分钟实行一次自动 build，build 结果将可视化地显示在 GitHub 中（图 5.16）。

图 5.16 Pull Request 的 build 结果



可以确认 Pull Request 的内容说明下方有“Determining merge status -- Merged build started.(Details)”这样的信息。这是为了判断是否能够合并而开始执行 build 时输出的消息[①]。点击 Details，会迁移到相关的 Jenkins 画面（图 5.17）。

图 5.17 在 Jenkins 中确认 Pull Request 的 build 结果



① 这时如果 build 已经成功，就会出现“Good to merge”；反之如果失败的话，则会出现“Failed”这样的消息。

从图 5.17 中可以看出，25 号的 Pull Request 造成了 7 个测试失败。也就是说，这个 Pull Request 不能合并到中央代码库。

这样的运用方法能够防止将造成 build 失败的提交合并到中央代码库。因此能够避免由于某人的提交有问题而造成全员作业停止这样的事态。

❻ 添加 White list 或重新执行 build

提交 Pull Request 的用户如果还没有被添加到 Admin list 或 White list 的话，build 则不会被执行。这时 bot 用户会在 Pull Request 中添加“Can one of the admins verify this patch ?”这样的评论。对此，Admin 将下列两个回答之中的任意一个作为评论添加在 Pull Request 中，即可通过 bot 用户指示插件开始执行 build。

```
ok to test
```

实施 build。

```
add to whitelist
```

将建立 Pull Request 的用户添加到 White list 并实施 build。这样，从下次开始，由该用户建立的 Pull Request 就会被自动 build。想要再次执行已经执行过的 build 时，只需添加下列评论即可 [①]。

```
test this please
```

如果觉得上述过程麻烦的话，只要在最初的任务配置阶段将用户添加到 White list 中即可。另外，正如 5.4 节的第一个专栏中所提到的那样，要想使用该插件，就需要支持 GitHub 向 Jenkins 发送请求的网络结构。添加只允许 GitHub 的 IP 地址发出请求的 IP 过滤规则，这样安全性方面就不用担心了。即使不使用 GitHub，也可以采用相同思路的运用策略。Jenkins 的作者川口耕介将这样的思路称为“验证后合并”[②]。只向主分支合并经过验证的、没有问题的修改，由此来避免因某人提交造成 build 失败，进而导致全体人员作业停止的事态，团队开发的生产效率也能更上一层楼。

① 这些命令语句自身都可以在图 5.15 的配置中修改。

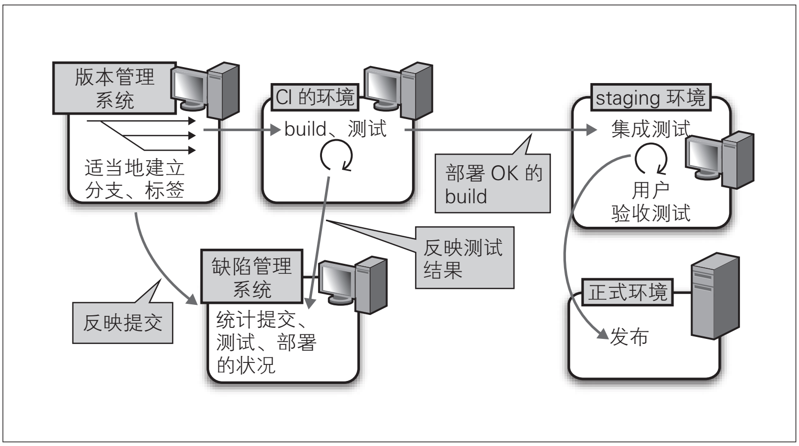
② 事实上 Subversion 也支持验证后合并的运用方式。

5.5.2 确保可追溯性

build 失败或发生问题时，有问题的 build 是哪次提交产生的，该提交原本是为了解决怎样的问题等，如果能够追溯上述信息，那么对于实际的运用是很有帮助的。

通过 Jenkins 和缺陷管理系统以及版本管理系统的协作，能够确保各类信息的可追溯性，实现项目的可视化（图 5.18）。

图 5.18 确保可追溯性

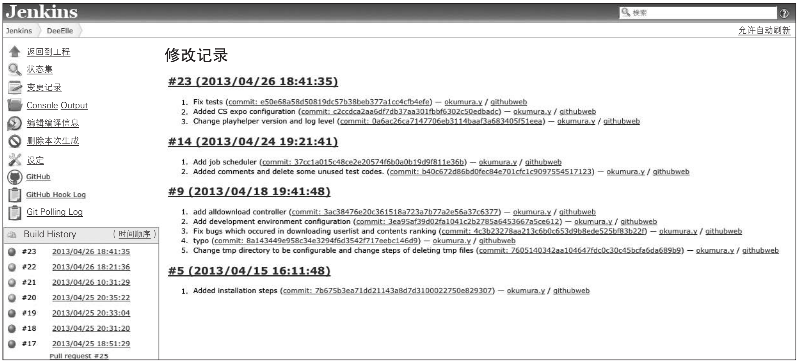


●……… 关联 build 和提交

阅读到这里并实施了 CI 的话，Jenkins 和版本管理系统的关联应该已经配置完成了[①]，当然不配置的话也无法实施 CI。也就是说，只要通过 Jenkins 实施 CI，就能将 build 和提交明确地关联起来。大致的印象如图 5.19 所示。

① 请参考 5.4.4 节的内容。

图 5.19 用 Jenkins 关联 build 和提交



如图 5.19 所示，在 Jenkins 的 build 号、Git 的提交编号以及提交者这些信息之间建立起了关联。例如，可以看出 #23build 和 3 个提交有关。另外，从图 5.19 中还可以看到有代码库浏览器（本例为 GitHub）的链接，可见 build 和代码的 Diff（差分）之间的关联也建立起来了。

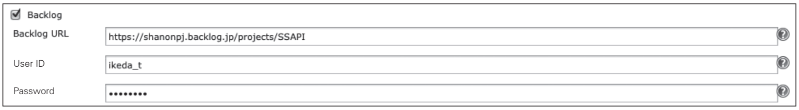
这样，即使 build 失败，也能够顺藤摸瓜地弄清是谁造成的、修改的内容是什么。这一功能在应对问题等时候是非常有用的。

●……… 关联缺陷管理

如果 Jenkins 已经和缺陷管理以及版本管理进行了关联的话，build 和提交，以及提交和相关问题票号之间就都建立起了相应的联系。

这次以 nulab 公司的 Backlog 产品为例进行说明。如同在第 4 章提到的，Backlog 是集缺陷管理系统和版本管理的代码库浏览器于一体的 SaaS 系统。想要关联 Jenkins 和 Backlog 的话，首先需要安装 Backlog 插件。和 Git 插件一样，请从管理插件处进行安装。安装后打开各任务的配置画面，会出现如下输入框，请输入 Backlog 工程的 URL、用户 ID（User ID）和密码（Password）（图 5.20）。同时在 Backlog 中也需要进行配置，将版本管理系统的提交记录和问题票关联起来。

图 5.20 Jenkins 和 Backlog 的关联



Backlog 能够和 Subversion 以及 Git 这两个版本管理系统进行协作。结合使用 Backlog 和 Subversion 的情况下，要选中图 5.21 中的“提交信息连接课题”选项。

图 5.21 Backlog 和 Subversion 的关联



Git 的话如图 5.22 所示。

图 5.22 Backlog 和 Git 的关联



这次使用了 Subversion 作为版本管理系统。至此配置就完成了。在这种状态下，将问题票号添加到提交记录中提交并执行 build，Jenkins 上就会生成如图 5.23 这样的修改记录画面。

图 5.23　在 Jenkins 上和 Backlog 的 Subversion 进行关联



点击 Backlog 的链接，迁移到如图 5.24 所示的代码库浏览器。这里能看到 Diff 等信息。

图 5.24　Backlog 的代码库浏览器



从代码库浏览器上可以确认问题票号及其链接。图 5.24 中的“SSAPI-21”部分就是问题票的链接，点击链接就能前往如图 5.25 所示的对应的题票。

图 5.25　能够链接到对应的问题票



像这样，Jenkins 提供了和 Backlog 等各种缺陷管理系统进行协作的插件。这次以 SaaS 形式提供的、便利的 Backlog 为例进行了说明。Trac 和 Redmine 也提供了 Jenkins 的插件，可以实现完全相同的配置。请以实现高可追溯性的 CI 为目标，有效地加以运用。

5.6 本章总结 借助 CI 能够实现的事情

本章介绍了 CI 的实践，并对实现 CI 所必需的编译工具、测试框架、CI 工具进行了说明。通过导入 CI，能够时常实施编译和测试，这样就能既迅速又准确地获得开发内容的反馈，在保持应用程序高质量的同时，还有助于提高开发速度。

另外，通过在 CI 服务器上对各类信息进行统一的管理，就既能确保可追溯性，又能够推进项目的可视化。借助 CI 还可以构筑和代码库完全对应的、能够时常保持完全运行状态的应用程序环境，比起临近发布时才开始提心吊胆地耗费长时间进行编译，这是很大的进步。

那么，在 CI 之后还有什么呢？借助 CI 我们已经确保了代码库中应用程序能够一直正常运行。而如果能将代码库中的程序直接发布到正式环境，那么应对市场变化的速度不就能大大提升了吗？

实现上述想法的实践就是持续交付（CD，Continuous Delivery）。第 6 章将介绍实现 CD 所必须要做的事情以及运用上的一些小技巧。

第6章

部署的自动化（持续交付）

6.1 应该如何部署

用一句话来概括这一章就是“所有部署相关的作业都应该实现自动化”。这里所说的部署是指将开发的代码以能够使用的状态放置到服务器上这一连串行为。把 WAR 文件上传到 Tomcat 等容器上的行为也可称为部署，但这个是更为广义上的部署。本章和《持续交付》[①]在内容上虽然有重复的部分，但本书更为简明易懂，笔者将根据自身过去数百次的部署经验，对部署的自动化进行简单的说明。重点对部署的最优方法以及简化部署的工具的使用方法进行讲解。

6.1.1　部署自动化带来的好处

实现部署的自动化会带来哪些改善？首先我们来说一下自动化部署的优点。

●……… 细粒度、频繁地发布可以使风险可控

部署工作本身就不是一件轻松的事情，如果几个月才能实施一次部署的话，程序就会有多个部分产生大量的代码修改。所有的修改都能正常运行自然最好，但现实往往并非如此。考虑到多个较大的故障同时发生所造成的严重后果，这的确是个棘手的问题。而如果实现部署的自动化和简易化，就能够频繁地实施部署，由此对故障规模进行控制就成为了可能。

●……… 能尽快地获得用户的反馈

部署得越早，就能获得越多的用户反馈。尽快让用户体验新开发的

① 《持续交付：发布可靠软件的系统方法》David Farley、Jez Humble 著，乔梁译，人民邮电出版社，2011

功能，并将用户的反馈反映到下一阶段的开发中，若能形成这样的良性循环，就能确立市场的优势地位。通过频繁地进行部署来回收开发的投资，才可能产生收益。

但上述情况的前提条件是：必须要有接受用户的反馈并反映到开发中的流程。具体来说包括分析用户的评论、意见以及系统的日志等，本章对此不做讲解。

●……… 团队的规模可控

如果有 10 个产品，采用 10 种不同的方法手动实施部署的话会怎么样？每个产品每月至少会有 1 次部署的工作，那么负责部署的运维团队就需要专门的人员来实施部署。如果运维团队的人手不足，就可能发生新产品无法部署的事态。而如果实现了自动化部署，就不必担心运维团队的人手不足，这样就能够推出新产品并获得用户反馈。借助部署自动化，团队人数的规模变得可控，因此可以放心地增加产品数量。

6.2 部署的自动化

要推进部署的自动化，就要解决这样一个问题：最初应该由谁着手实施？如何实施？其实答案就是“由想实施部署自动化的人着手去做就行了”。虽然有些简单粗暴，但这是最合理的解答。如果可能的话，最好以对服务器及网络相关事情有决定权的运维团队人员作为核心成员。因为不可避免地要预先做好环境相关的准备及调整，以及最终向正式环境实施自动化部署所需的准备工作。因此由运维团队的人员来牵头着手部署自动化，项目的进展会比较顺利。

6.2.1　部署自动化方面的共识

支撑部署自动化的技术方面，要选用开发人员（Dev）和运维人员（Ops）都能够使用的技术。关于这一点，在充分协商的基础上，双方达成一致是非常重要的。缺少任何一方的协助，都无法顺利实现部署的自动化。因为部署是指从开发到向正式环境发布应用程序为止的一连串行为。

要实现部署的自动化，需要全体团队成员就下面列举的 4 个项目达成统一认识。

❶ 要全部采用版本管理

❷ 所有的环境都要用同样的方式来构建

❸ 要实现发布工作的自动化，并事先进行验证

❹ 要反复多次进行测试

❶ 的话请参考第 3 章的版本管理系统。不仅仅是应用程序的代码，自动化相关的工具、数据库、测试用的数据等都应该作为版本管理的对象。这么做是为了在任何时候都能够恢复到过去的任意状态。

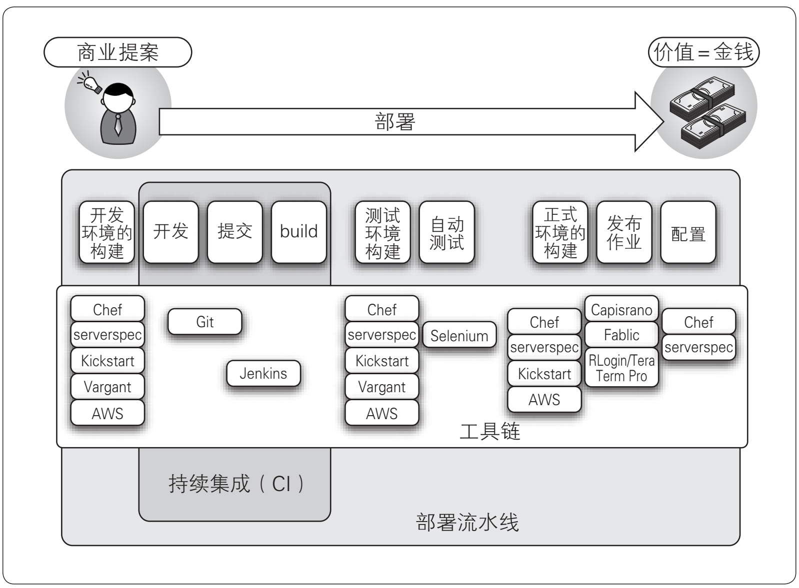
关于 ❷ 和 ❸，我们要意识到应该采用相同的方法来构筑开发环境、测试环境、正式环境并实施发布。这是在推进使用工具进行环境构建的过程中必须努力做到的要点之一。即便是手动修改了 1 行配置，那么从这一刻起这个环境中就混入了只有实施修改者才知道的信息。

❹ 是指对自动化部署实施持续的验证，大量收集其成功和失败的信息。只有这样，向正式环境实施部署工作的准确性才能得到保证。如果能够做到上述全部内容，那么部署将不再是可怕的事情。

6.2.2　部署流水线

实现应用程序的 build、部署、测试、发布这一系列流程的自动化称为部署流水线（deployment pipeline）[①]（图 6.1）。

图 6.1　部署流水线



① 详细内容请参考之前提到的《持续交付》。

●········ 通过自动化加快部署速度

我们的目标是构建这样的部署流水线，并通过流水线的循环尽量加快部署的速度，同时还必须确保正确性。能够想到的最差情况就是：从着手开发到发布经历了数月～数年之久，并且全部采用手工作业，还出了差错。因此，在部署流水线的每一个阶段都需要全体团队成员就之前提到的部署自动化中的 4 个项目达成统一认识。

部署流水线的每个阶段都有着各种可以使用的工具。将各个阶段的工具组合起来就有可能加快部署的速度。

例如，提交某个 bug 的修改，由提交引发持续集成的运行，顺利通过 build 后自动部署到测试环境，执行用户验收测试，通过测试的话向正式环境实施部署，在正式环境上实施完冒烟测试[①]后向相关人员发送通知。到此为止的所有流程全部以自动化的形式实施。上述流程虽说是比较极端的例子，但确实是一种理想形式。

●········ 任何人都能够实施部署是很重要的

实际上一般的流程是当提交积累到一定数量时建立分支，并向正式环境实施部署。这里的重点是部署流水线要能够顺畅地运转起来，在部署的各个阶段都不允许有因为对人的依赖而无法部署的情况。理想的情况是任何人都可以实施测试，也都可以实施发布。

当然考虑到访问权限和审批流程的问题，最后按下发布按键的可能是特定的人，但并不是说这需要什么特别的技术。换而言之就是：不需要“发布技师”这样的人。

6.2.3 服务提供工具链（provisioning tool chain）

再稍微详细地对图 6.1 中给出的工具链进行一下说明。大致可以分为以下 3 个层次来考虑（图 6.2）。

- **引导（Bootstrapping）... 服务器 OS 的配置及基于虚拟机的服务**

① 修改代码进行 build 时，为了确认 build 是否正常结束而进行的测试。

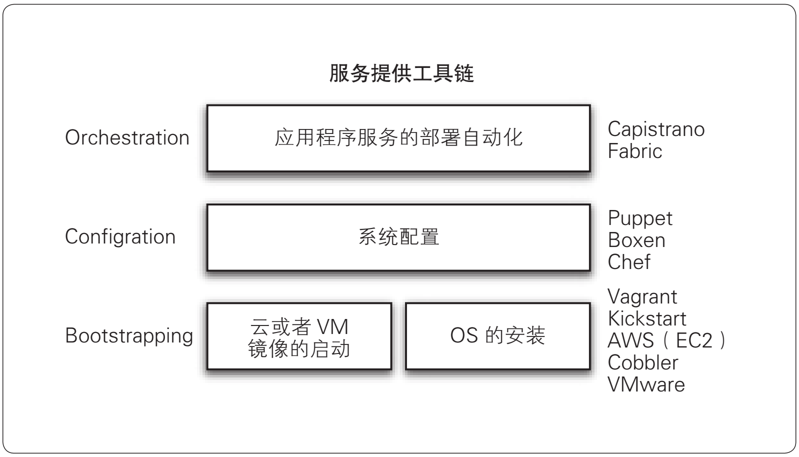
器安装自动化的相关工具

- 配置（Configuration）... 服务器及中间件的配置自动化工具
- 业务流程（Orchestration）... 代码部署及发布相关的服务器操作等自动化工具

以上 3 个层次都无法只借助 1 个工具完成所有处理，只有集齐各个层次的工具并实现流水作业，部署流水线才能够完成。

接着我们来介绍几款在各层次中出现的工具。

图 6.2 服务提供工具链



6.3 引导（Bootstrapping）

引导层启动服务器的 OS，使服务器达到我们所需要的状态。这个层次包括了 OS 的种类、磁盘容量以及网络配置等。本节我们将介绍利用服务器自动化安装以及虚拟化技术来构建环境的自动化工具。

6.3.1 Kickstart

假如你所在的开发现场添置了 100 台服务器。如果所有的服务器都必须设定同样的磁盘分区、选择初始化安装包、添加账户以及设置密码，你应该怎么做?

虽然也可以为 100 台服务器插入安装 CD，接上键盘，一台一台地配置磁盘分区，但这样会耗费大量的时间。在这种情况下，比较有用的工具就是 Kickstart。

●……… Kickstart 的使用方法

Kickstart 的使用方法是：安装 Linux 时，在 kernel 参数中加上 [ks=...] 这样的选项，就可以从 USB、外置硬盘等外部存储器或 HTTP、FTP 站点加载配置文件以实现安装自动化。

结合 PXE[①]启动，只要在插着网线和电源线的状态下按下电源开关，即可自动完成安装。并且不仅限于物理服务器，Kickstart 同样也可用于虚拟机的安装，因此通用性非常好。

●……… 使用时的注意事项

需要注意的是，Kickstart 只能用于 Red Hat Enterprise Linux 系列（以下简称 RHEL 系列）Linux 发布版本的安装。Debian GNU/Linux 系

① Preboot Execution Environment，网络启动规格的一种。

列的 Preseed、Solaris 需要使用 JumpStart 这种其他的安装形式。

有时还需要根据磁盘容量等硬件规格的不同来设置不同的分区方式。在引导层还需要意识到：根据所用的工具以及内容的不同，可以选择的硬件及 OS 等也是有所差异的。

●……… Kickstart 的配置示例

下列代码是在连接网络的状态下，安装并启动服务器的简单配置示例。Kickstart 的内容是将 CD 中安装向导所问的问题以配置文件的形式确定下来。

```
# Kickstart file automatically generated by anaconda.

install
url --url=ftp://ftp.jaist.ac.jp/pub/Linux/CentOS/6/os/x86_64/
    ↑指定通过FTP来安装
lang ja_JP.UTF-8
network --bootproto dhcp
keyboard jp106
zerombr
clearpart --all
part / --fstype ext4 --size=1 --grow --asprimary
    ↑分区配置的描述。设置为grow的话会将剩余所有的磁盘空间都分配给该分区
part /var --fstype ext4 --size=40960
part /home --fstype ext4 --size=81920
part /boot --fstype ext4 --size=400
part swap --size=4096
bootloader --location=mbr --driveorder=sda --append="crashkernel=auto rhgb
quiet"
timezone --utc Asia/Tokyo
rootpw password ←设置root的初始密码
user --name=defaultuser --groups=users --password=userpass
    ↑设置首次登陆的用户
selinux --disabled
firewall --disabled
authconfig --enableshadow --passalgo=sha512
reboot ←安装完成后重启
repo --name="CentOS"  --baseurl=ftp://ftp.jaist.ac.jp/pub/Linux/CentOS/6/
os/x86_64/ --cost=100

%packages   ←指定安装包的顺序
@base
@console-internet
@core
```

```
@debugging
@japanese-support
@large-systems
@network-file-system-client
@server-platform
@server-policy
pax
oddjob
sgpio
certmonger
pam_krb5
krb5-workstation

#setup start      ←包安装完成后运行shell脚本
%post --log=/tmp/anaconda-post.log #--erroronfail
function log_mesg() {
/bin/echo
/bin/echo %POST: $*
}

log_mesg "Start server setup"
yum -y update

log_mesg "End server setup"
/sbin/ifconfig | /bin/mail -s "Complete Server installation." user_email_
address@example.com
log_mesg "END %POST SCRIPT"
```

可以在 part 部分记载分区配置，在 user 部分建立初始用户并设置密码。还可以配置初始安装的软件包，并且在安装完成后自由地执行 shell 脚本。这里用 `yum`命令将包更新到最新状态，最后通过 `mail`命令通知 `ifconfig`命令的执行结果。

6.3.2　Vagrant

开发人员想试着运行下最新代码的效果时，应该准备怎样的环境才好呢？

●……… 为每一位开发人员准备实体电脑比较困难

有了 Kickstart 这样的机制，为实体电脑安装 OS 已经变得比较简单，但如果为每一位开发人员准备一台专用服务器的话，成本会非常

高。另外还有物理上的限制，诸如没有放置的空间或供电不足等问题，因此为每一位开发人员提供一台服务器是并不太现实的。

●……… 使用虚拟机时的注意事项

解决上述问题的办法之一就是将测试环境构建在虚拟机上，但虚拟机的运用、操作多少需要具备特殊的知识，有时可能无法顺利地使用虚拟机。

在还不能方便地使用虚拟机的情况下，可以在和 staging 环境以及手头的开发环境大致相同的运行环境下 checkout 最新的代码，试着确认程序的动作。但各个环境上应用程序所依赖的模块的安装版本存在差异，无法进行验证的情况也是常有的。

轻松地搭建虚拟机，在不影响当前使用环境的前提下就能够进行尝试，安装所需要的模块并实施验证。下面就来介绍一下能够实现上述功能的软件 Vagrant。

●……… 什么是 Vagrant

Vagrant[①]主要是通过 CLI（Command Line Interface）来操作名为 VirtualBox[②]的虚拟机软件的工具，由 Ruby 编写。这里说“主要”，是因为 Vagrant 还支持 VMware fusion 的操作，并且通过安装插件，还可以实现类似于 AWS 的 IaaS 虚拟机的操作。

Vagrant 不需要特殊的配置，也无需通过鼠标进行复杂的操作。VirtualBox 的运行环境是多种平台的，Windows、Mac、Linux 等只要是 x86 架构的 OS 都可以运行，因此能在多种 OS 上使用。

●……… Vagrant 的安装及运行方法

旧版本（1.1 版以前）的 Vagrant 采用 `gem` 命令的形式来安装。最新版本以 RPM 或 DEB 包的形式提供，因此可以从官网[③]下载并安装，

---

① http://www.vagrantup.com/

② https://www.virtualbox.org/

③ http://www.vagrantup.com/

安装完成后即可开始使用。安装包中除了 Vagrant 的代码之外，还包括 Ruby 的本体等，因此不会和安装对象机器上安装的 Ruby 版本发生冲突。无论在运行有怎样的应用程序的环境上都能方便地运行。

首先就让我们访问 Vagrant 的主页，下载最新的包并安装。安装完成后，从 VagrantBox 支持的 OS 模板列表[①]确认镜像的 URL，并启动虚拟机。

```
$ vagrant box add centos http://developer.nrel.gov/downloads/vagrant-
boxes/CentOS-6.3-x86_64-v20130101.box
$ vagrant init centos
$ vagrant up
```

只需这几步操作就能在本地机器上构筑起虚拟的 CentOS 环境。执行`vagrant box add`命令的部分就是将 OS 的镜像加载到 Vagrant 中。`vagrant box add`支持多种镜像，可以通过运行 `vagrant box list`命令来确认所支持的镜像列表。本例中设置了 CentOS 镜像的 URL，其他如 Ubuntu、BSD 的镜像等也可以在网上找到。

接着让我们试着登录到启动后的虚拟机环境。

```
$ vagrant ssh
```

Mac 和 Linux 环境下可以直接登录。Windows 环境因为没有安装 `ssh`命令，所以无法通过 `vagrant ssh`命令来进行。

在 Windows 环境下登录虚拟机的方法有通过 Tera Term[②]等终端进行 SSH 登录，或者在 Cygwin 环境下安装 `ssh`命令，通过执行 `vagrant ssh`命令进行登录。

登录后可以对环境进行各类验证，如果不再需要该镜像或者验证失败、环境损坏的话，能够直接删除虚拟机。

```
$ vagrant halt
$ vagrant destroy
```

像这样，虚拟机的关闭以及删除也可以通过命令轻松地实现。如果还想在新的干净的环境上进行尝试的话，只需执行 `vagrant up`命令，就能建立新的虚拟机环境，这样就可以反复地进行尝试。

① http://www.vagrantbox.es/
② http://sourceforge.jp/projects/ttssh2/

因为 Vagrant 是由 Ruby 编写的，所以和稍后介绍的 Configuration 工具 Chef 的配合度很高。结合使用 Vagrant 和 Chef，能够反复地实施环境的构建及服务器、中间件的配置的验证。比起在本地环境上实施上述处理，建议准备好实施 CI 的专用环境，具体的内容将稍后讲解。

6.4 配置（Configuration）

6.4.1 不使用自动化时的问题

说起部署，人们往往只关注发布工作，但其实服务器的构成管理成本也非常高，这一直是一个问题。例如，某应用程序的运行环境非常复杂，构建该环境需要经过数十、数百个步骤的操作，若在这样的情况下搭建新的验证环境或开发环境，会有怎样的问题呢？

笔者所经历过的某个应用程序开发中，由于没有人负责维护构建环境的手册，只提供了正常运行环境的访问权限，因此此只能一边确认此环境上安装的模块及中间件的配置，一边构建手头的环境，白白浪费了时间和精力。

利用虚拟化技术，充分利用作为原型的虚拟镜像，的确能多少跳过一些中间步骤，但即使是习惯的人也要花费几小时，不习惯的人甚至要花费几天的时间。在实际的开发工作中，当团队人员增加或发生工作交接等情况时，构建环境以外的人也必须能够在该环境上运行应用程序。

耗费了一定的时间终于构建起了大致版本一致或相同的运行环境，程序也开始运行了。但这并不意味着各个环境之间就绝对一致。

若在这样的环境下进行开发，就会出现“明明在本地环境上能够正常运行，但在测试环境下就运行不了”的问题。正是因为模块间微妙的版本差异而产生了 bug。

作为上述问题的解决方案，可以使用配置（Configuration）相关的自动化工具，从一开始就采用完全相同的方法对环境进行安装。

6.4.2 Chef

Chef 是用 Ruby 编写的服务器配置的自动化工具。只要准备好名为 Cookbooks（食谱）的服务器构建手册形式的配置文件等，就能按照文件所记述的规则为服务器安装软件包并配置中间件。

无论 10 台还是 1 万台机器，只需要 1 份 Cookbooks，就能构筑起符合要求的环境。像 Facebook 这样大规模的运维也同样使用 Chef[①]。

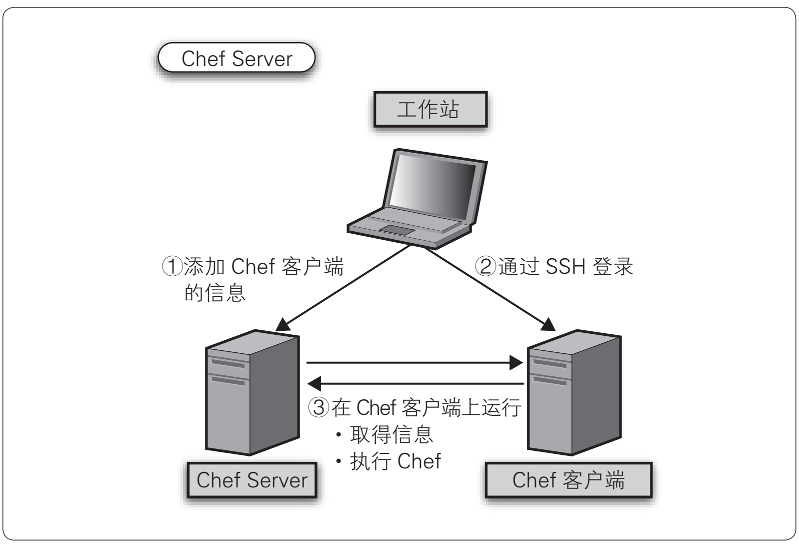
Chef 不仅限于大规模基础设施的构建，在小规模基础设施构建中也能发挥其真正的价值，因此即便只是构建 1 台机器的环境，也可以对其加以有效利用。

●········ Chef 的构成

Chef 的运行大致有以下 3 种结构。

❶ Chef Server（图 6.3）

图 6.3 Chef Server



① http://www.infoq.com/cn/news/2013/02/facebook-chef

❷ Chef Solo+ Knife Solo（图 6.4）

❸ Chef Solo（图 6.5）

图 6.4 Chef Solo + Knife Solo

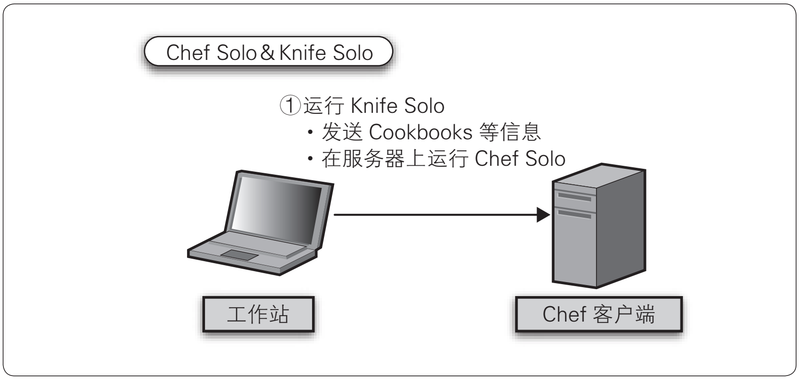
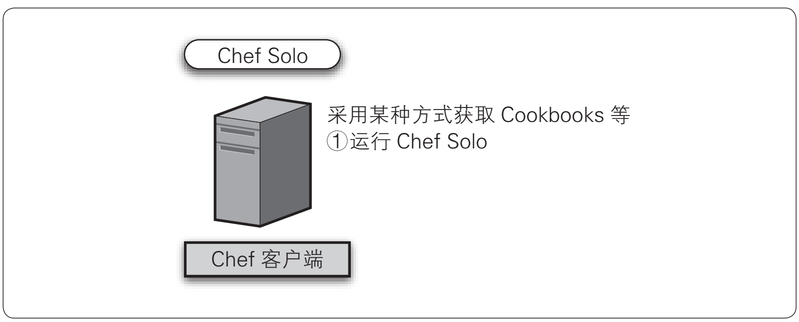


图 6.5 Chef Solo



❶～❸ 所使用的 Cookbooks 可以是相同的。在由 1～2 台服务器构成的环境下，可以从 Chef Solo 的构成方式开始，然后随着服务器台数的增加，再考虑构建 Chef Server 的环境。

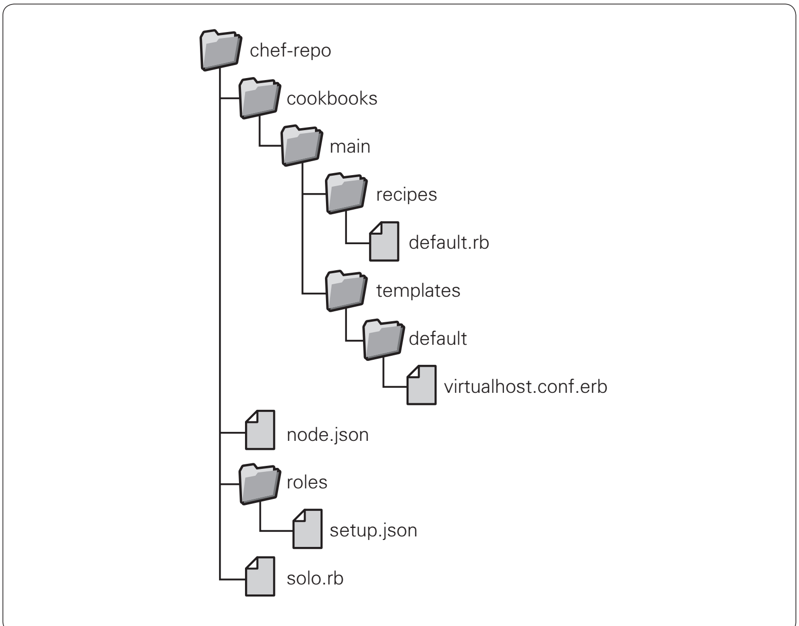
为了学习 Chef 最基本的使用方法，这里我们以利用 Chef Solo 构建简单的 Web 服务器环境为例进行讲解。

Chef 的安装和 Vagrant 一样可以采用 RPM 等包的形式进行。过去是通过 gem 命令进行安装的，现在安装包中还提供了 Ruby 的本体，因此不会和 OS 上的 Ruby 发生版本依赖等问题，能够直接使用 Chef。

●········ 目录构成和文件配置

这次介绍的 Chef 的文件和目录结构如图 6.6 所示。Chef 的配置文件统称为 Cookbooks，其中特别是 Recipe 和 Template 起着主要的作用。下面就对这些文件逐个地进行讲解。

图 6.6 Chef 的文件和目录结构



文件和目录可以用 Chef 的命令行接口 Knife 自动生成。

```
$ knife cookbook create <Cookbook name> -o <output dir>
```

这样就能自动生成必要的目录结构和文件，所以在第一次构建 Chef 环境时，请试着执行上述命令。但是指定执行对象 Recipe 的 node.json 等配置文件需要手动生成。

●········ node.json

这个文件是运行 Chef 所必需的文件。这次 node.json 中记载的内容参照了 roles 目录下的 setup.json。本例中记述的是执行 main 目录下的名

为 default.rb 的 Recipe。

```
{
  "run_list": [
    "role[setup]"    ←参照setup.json
  ]
}
```

●……… setup.json

这个文件就是配置实际的 Recipe 文件的地方，同时还记载了 attribute。attribute 定义的变量能在 Recipe 文件和 Template 文件中使用。

```
{
 "name": "setup_examplep_program",
 "override_attributes": {
     "apache": {
       "documentroot": "/var/www"     ←能够在recipe或template中使用的变量
     }
 },
 "json_class": "Chef::Role",
 "description": "Setup Examplep Program",
 "chef_type": "role",
 "run_list": [
               "recipe[main::default]"    ←配置 Recipe
             ]
}
```

●……… solo.rb

这个文件是运行 Chef 所必需的文件。记载有 Chef 的 Cookbooks 的路径以及 roles 的文件路径。Chef 是用 Ruby 编写的，所以路径的配置同样可以使用 Ruby 的 File.expand_path。

其他 Ruby 的语法也可以使用，但原则上作为服务器构建的自动化工具应该尽量将所使用的语言限制在 Chef 的 DSL（Domain Specific Language，领域语言）范围内。

```
cookbook_path File.expand_path("../cookbooks", __FILE__)
role_path File.expand_path("../roles", __FILE__)
```

●⋯⋯ default.rb

这个文件就是被称为 Recipe 的重要的配置文件。在 package 那一行中，RHEL 系列的话使用 yum/rpm，Debian GNU/Linux 系列的话使用 apt/deb 来安装 Apache。

template 是在部署配置文件时使用的，这里记述的处理是在 Apache 的 conf.d 目录下添加新的配置文件。variables 用于在 Recipe 中接收 template 的变量。Chef 中对变量的处理是非常重要的。通过记述在 attribute 中，就可以在 Recipe 和 template 中作为变量使用，但 attribute 可以记载在几个文件中，并且有着各自不同的优先级。本书不做详细的介绍，只是提一下优先级最低的 attribute/default.rb 文件中所记载的是作为全体的默认值，关于服务器的作用、测试环境、正式环境这样不同环境的环境变量，可以记载在 Role 或 Environments 中。通过灵活运用 attribute，能够调整单个 Cookbooks 使其适用于各种环境。

在 directory 那行中，虽然只定义了新建指定的目录，但需要注意的是这里用到了定义在 attribute 中的 `#{node[:apache][:documentroot]}` 这样的变量。

git 那行记述了 checkout 代码库的 master 分支到指定的目录下。通过指定 `action :sync`，checkout 时就会更新最新的版本。

在 service 那行中，可以操作中间件的启动或停止。这里定义为启动 Apache。

```
package "httpd"   ←安装httpd

template "/etc/httpd/conf.d/#{ENV['HOSTNAME']}.conf" do
  ↑根据template部署文件
    source "virtualhost.conf.erb"
    mode 0644
    owner "root"
    group "root"
    variables({
        :env_hostname    => "#{ENV['HOSTNAME']}",
    })
end

directory "#{node[:apache][:documentroot]}/#{ENV['HOSTNAME']}" do
    action :create   ←建立由attribute指定的目录
```

```
end

git "/var/www/#{ENV['HOSTNAME']}" do
  repository "https://github.com/example/program.git
   ↑指定任意程序的Git仓库
  reference "master"
  action :sync
end

service "httpd" do
    action :start  ←启动httpd
end
```

virtualhost.conf.erb

这是 Recipe 中指定的 Template 文件。这里可以内嵌 attribute 或 Recipe 文件中以 variables 形式指定的变量。这里可以使用 Ruby 的 erb 模板。

```
<VirtualHost *:80>
    ServerAdmin serveradmin@<%= @env_hostname %>
    DocumentRoot <%= @node.apache.documentroot %>/<%= @env_hostname %>
    ServerName <%= @env_hostname %>
    ErrorLog logs/<%= @env_hostname %>-error_log
    CustomLog logs/<%= @env_hostname %>-access_log combined
</VirtualHost>
```

Chef 的运行方法和运行结果

只需指定 node.json 和 solo.rb，就可以运行 Chef Solo。

```
$ sudo chef-solo -j node.json -c solo.rb
```

执行 Chef 后显示如下结果。

```
[root@ip-10-132-183-178 chef]# chef-solo -j node.json -c solo.rb
Starting Chef Client, version 11.6.0
Compiling Cookbooks...
Converging 5 resources
Recipe: main::default
  * package[httpd] action install   ←安装httpd
    - install version 2.2.25-1.0.amzn1 of package httpd

  * template[/etc/httpd/conf.d/ip-10-132-183-178.conf] action create
    - create new file /etc/httpd/conf.d/ip-10-132-183-178.conf
```

```
    - update content in file /etc/httpd/conf.d/ip-10-132-183-178.conf from
 none to 3cb554  ←部署Template中指定的文件
        --- /etc/httpd/conf.d/ip-10-132-183-178.conf    2013-09-22
10:12:45.278335727 +0000
        +++ /tmp/chef-rendered-template20130922-6866-6hkxkf-0   2013-09-22
 10:12:45.290335967 +0000
        @@ -0,0 +1,8 @@
        +<VirtualHost *:80>
        +    ServerAdmin serveradmin@ip-10-132-183-178
        +    DocumentRoot /var/www/ip-10-132-183-178
        +    ServerName ip-10-132-183-178
        +    ErrorLog logs/ip-10-132-183-178-error_log
        +    CustomLog logs/ip-10-132-183-178-access_log combined
        +</VirtualHost>    ↑文件的内容被替换成了attribute的值
        +
    - change mode from '' to '0644'
    - change owner from '' to 'root'
    - change group from '' to 'root'

  * directory[/var/www/ip-10-132-183-178] action create
    - create new directory /var/www/ip-10-132-183-178
      ↑建立了由attribute指定的目录
  * git[/var/www/ip-10-132-183-178] action sync
    - clone from https://github.com/example/program.git into /var/www/
ip-10-132-183-178    ←使用git命令进行clone
    - checkout ref 05e45129b1fd1bbad18c790fcd814ba36403cab1 branch master

  * service[httpd] action start    ←启动httpd
    - start service service[httpd]

Chef Client finished, 5 resources updated
```

使用这个 Cookbooks 安装 Apache，并将服务器配置为基于名称的虚拟主机，在 DocumentRoot 中通过 Git checkout 任意程序并启动 Apache。到此为止的处理就实现自动化了。如果是 checkout 后能够立即使用的应用程序的话，那么通过上述一连串的处理，环境构筑应该就能完成了。

●……… 使用 Chef 的优点

可以看出用 Chef 的 DSL 编写的 Cookbooks 非常容易理解，学习成本非常低。Cookbooks 是用 Ruby 的代码编写的，因此还可以进行版本管理，即相当于在版本管理系统上保存了“服务器正常状态及其历史记录”。

至今为止服务器的结构和构建管理是被分割开的，通过对 Cookbooks 进行版本管理，服务器的构建有了历史记录管理。并且只需执行 Chef 命令就能实现服务器构建的自动化。也就是说，用 Ruby 的代码来管理服务器状态的“Infrastructure as Code”这种服务器管理的新时代已经到来了。

●……… 使用 Chef 时的注意事项

使用 Chef 时有一点需要注意，那就是既然已经使用 Chef 对服务器进行管理，就要禁止手动构建服务器，要记住必须使用 Chef 进行相关作业。如果手动地安装包、修改配置等，就会造成服务器的状态和 Cookbooks 中的定义发生背离，而为了达到相同的状态就不得不执行多条命令，并且只有进行此操作的人才知道具体的内容。但如果将上述操作记载到 Cookbooks 的话，那么谁都可以通过 1 条 Chef 命令将服务器配置到相同的状态。

从构建管理的角度来看，不使用 Cookbooks 而手动构建服务器也是不明智的。全部采用 Chef 来构建环境，即使最初只需要 1 条命令就能完成构建工作，编写 Recipe 文件也是必要的。虽然对编写 Recipe 的工作感到焦躁不安的人不在少数，但经过 1 个月、1 年的时间后，随着配置修改的内容不断积累，你就会感受到使用 Chef 所带来的好处。

●……… 使用 Chef 的时间点

可以在构建开发环境的阶段，也就是开始开发前后一段时间开始使用 Chef。使用和开发环境、测试环境、正式环境同样的 Cookbooks，因环境而有所差异的配置，可以全部通过 attribute 来控制。

如果想进一步了解 Chef 的话，可以参考相关资料。

6.4.3 serverspec

●……… 什么是 serverspec

如果说 Chef 是用代码来管理服务器环境构建的工具，那么 serverspec 就是对服务器的构成进行单元测试的测试框架。从名字就可以看出，其内部使用了 Ruby 的 BDD 框架 RSpec①。

serverspec 虽然不是直接进行配置的工具，但为了确保 Chef 等通过代码管理服务器构建的工具能够持续地正常运行，serverspec 可以说是必不可少的测试框架。

●……… serverspec 的安装

执行如下命令来安装 serverspec 的运行环境。

```
$ gem install serverspec
$ serverspec-init
Select a backend type:

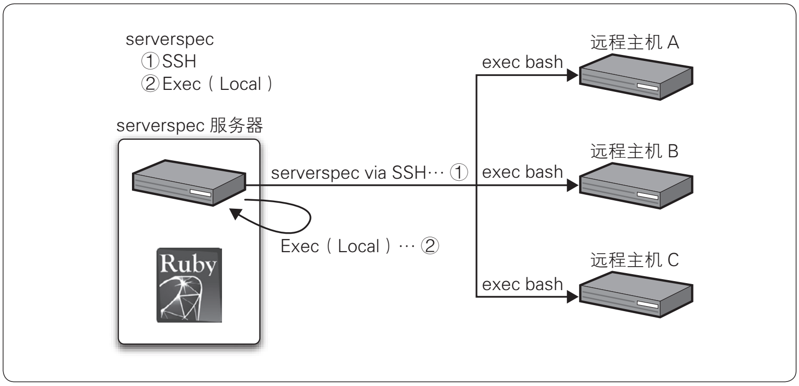
  1) SSH
  2) Exec (local)

Select number: 2
```

serverspec 支持远程服务器的测试，当然也可用于测试本地服务器（图 6.7）。执行测试需要 root 权限，或者通过 sudo 来执行。第一次初始化时无论选择远程主机（SSH）还是本地主机（Exec (local)），之后都可以进行切换，以验证为目的的话可以选择“2”，并制作测试用例。

① http://rspec.info/

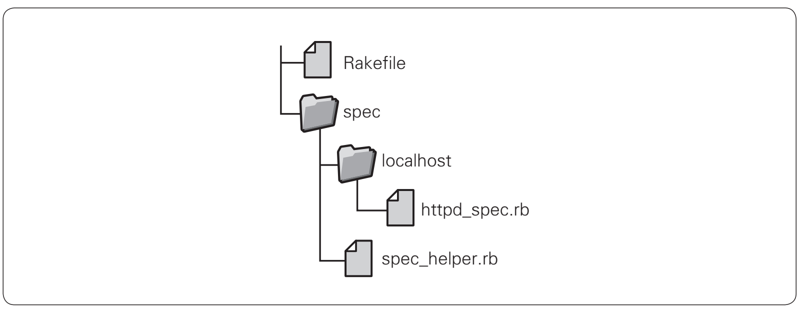
图 6.7　serverspec 的运行环境



●……… 测试文件的记述方式

执行 `serverspec-init`命令，会自动生成文件和目录（图 6.8）。

图 6.8　serverspec 的目录结构



localhost 目录下的 xxxx_spec.rb 文件就是测试用例。假设在执行之前提到的 Chef 的 Cookbooks 后进行测试，我们来试着制作测试用例。测试的内容包括确认 httpd 的安装及启动、确认通过 template 部署的文件是否存在，以及对文件内容的检查。

●……… httpd_spec.rb

```
require 'spec_helper'

describe package('httpd') do
```

```
  it { should be_installed }     ←检查httpd的安装
end

describe service('httpd') do
  it { should be_running   }     ←检查httpd是否启动
end

describe port(80) do
  it { should be_listening }     ←检查是否在侦听80端口
end

check_config = "/etc/httpd/conf.d/#{ENV['HOSTNAME']}.conf"
describe file(check_config) do
  it { should be_file }     ←检查通过template部署的文件的有无

  it { should contain "ServerName #{ENV['HOSTNAME']}" }
end     ↑检查文件的内容

describe file(check_config) do
  it { should be_mode 644 }
  it { should be_owned_by 'root' }
  it { should be_grouped_into 'root' }
   ↑检查文件的权限以及所有者
end
```

除了 httpd 以外，我们再试着添加`git clone`是否成功执行的测试用例。新建名为 spec/localhost/git_spec.rb 的文件，如下所示。

●········ git_spec.rb

```
require 'spec_helper'

describe file("/var/www/#{ENV['HOSTNAME']}") do
  it { should be_directory }
end     ↑检查git clone是否成功执行
```

●········ serverspec 的执行方法及执行结果

执行 serverspec 的测试时，在 Rakefile 层执行下面的命令。

```
$ rake spec
```

执行后会显示如下结果。

```
/usr/bin/ruby -S rspec spec/localhost/git_spec.rb spec/localhost/httpd_
spec.rb
```

```
……

Finished in 0.21744 seconds
9 examples, 0 failures
```

在上述例子中，可以确认 9 个测试用例都通过了。测试失败的话，failures 的数目会增加。如果要输出测试的详细内容，可以设置 `SPEC_OPTS="--format=documentation--colour"`这样的环境变量，就能输出 RSpec 的 documentation format。

要注意的是几乎所有 serverspec 的测试用例都被描述为了“should be_xxxxx”，一看便知这是在对服务器当前的状态进行检查。正因为可以像这样以近似自然语言的形式来描述测试用例，所以测试用例还可以被当作服务器的需求规格书来使用，制作非常简单。

●……… serverspec 的优点

serverspec 的特别优秀之处在于测试的执行不依赖于 Ruby 的运行环境这一点。测试代码的执行是通过 shell 命令进行的，因此只需要在运行 serverspec 的服务器上安装 Ruby 和 serverspec，远程服务器只需要能够通过 SSH 登录即可进行测试。

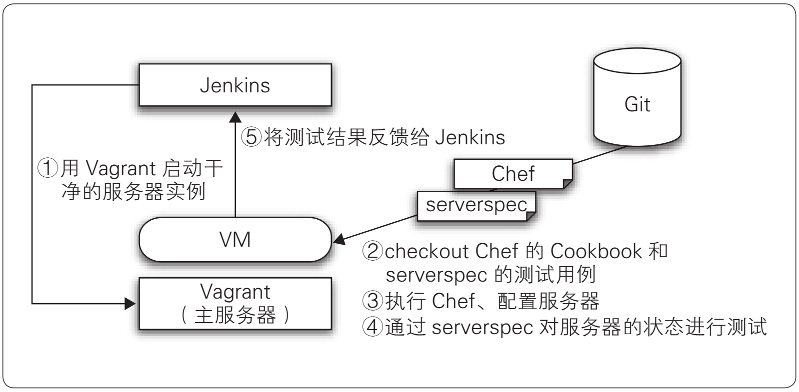
serverspec 还支持多种 OS，可以对 RHEL 以外的 OS 如 Solaris、Debian GNU/Linux 进行测试。

使用 serverspec 能够对任意环境应有的状态进行检查，配合 Chef 一起使用，就能够放心地对服务器进行动态的配置修改。

6.4.4 最佳实践（其 1）

在讲解 Chef 时提到了“Infrastructure as Code”这个单词，为了保证代码的质量，进行测试是很重要的，并且测试还必须能够持续地执行。在实现了服务器构建的自动化之后，CI 的实施就变得非常重要（图 6.9）。

图 6.9 Jenkins+Chef+serverspec+Vagrant



让我们试着结合 Jenkins+Chef+serverspec+Vagrant 来实际进行上述处理。

Vagrant 提供了启动虚拟服务器时执行 Chef 或 shell 的机制，这使得构建理想中的环境成为可能。执行 `vagrant init` 时会生成名为 Vagrantfile 的虚拟机配置文件。

在 Vagrantfile 中添加 Chef Solo 或 shell 脚本等服务器预处理（provisioning）的相关内容，在第一次执行 `vagrant up` 时，这些处理会被执行。除第一次启动之外，可以通过启动时添加选项 `vagrant up --provision`，或者在虚拟服务器启动后运行 `vagrant provision` 来执行上述预处理。

在 GitHub 上设有 Chef 或 serverspec 的代码库的情况下，通过配置 Webhooks，并设置在执行 Vagrant 的预处理的过程中启动 Jenkins 的任务，这样 CI 环境就构建完成了。git 代码库的话可以使用 git hooks[①]，几乎所有的代码库都支持在 commit 或 push 的钩子中执行命令。

Vagrantfile 中可以通过关键字 `inline:` 来提示 shell 脚本。下面的 Vagrantfile 会执行 `git clone` 命令。

① http://git-scm.com/book/zh/v1/ 自定义 -Git-Git 挂钩

Vagrantfile

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

# Vagrantfile API/syntax version. Don't touch unless you know what you're
doing!
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "centos"

  config.vm.provision "shell" ,     ←chef的安装
    inline: "curl -L https://www.opscode.com/chef/install.sh | bash"

  config.vm.provision "chef_solo" do |chef|
    chef.add_recipe "main"
    ↑设置安装git和serverspec的Recipe文件
  end

end

config.vm.provision "shell" ,     ←clone Cookbooks和测试用例
    inline: "rm -rf cookbooks_serverspec && git clone git@github.com/my/
cookbooks_serverspec.git"

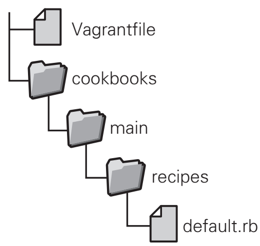
config.vm.provision "shell" ,     ←执行clone下来的Cookbooks
    inline: "cd cookbooks_serverspec/chef-repo  &&  chef-solo -j node.json
-c solo.rb"

config.vm.provision "shell" ,     ←最后执行serverspec
    inline: "cd cookbooks_serverspec/serverspec &&  rake spec"


end
```

通过指定 `add_recipe` 在主服务器上准备好 Recipe 文件，就会向虚拟服务器自动部署 Recipe 文件并执行 Chef Solo（图 6.10）。还可以通过指定 `cookbooks_path` 来使用定制过的 Cookbooks。

图 6.10 指定 add_recipe 的情况下的目录结构



由于 Vagrant 本体中没有提供 serverspec，因此通过 `inline:`来执行。或者也可以使用名为 `vagrant-serverspec`的插件。

●········ default.rb

```
package "git"
package "ruby"
package "rubygems"
gem_package "serverspec" do
   action :install
end
```

这只是服务器构建 CI 的一个例子。将 Vagrant 部分替换为 Docker[①] 等技术也可以实现相同的功能，Chef 部分也可用 puppet[②]来代替。serverspec 部分也可以用 Cucumber Chef[③]或 Test Kitchen [④]等测试框架。这里的重点在于要选择团队最易于使用的、适合自己的工具，而不是局限于本书中所提到的工具。这对于持续地运行这些工具来说也是非常重要的，因此工具的选择可以在慎重地交流沟通后再决定。

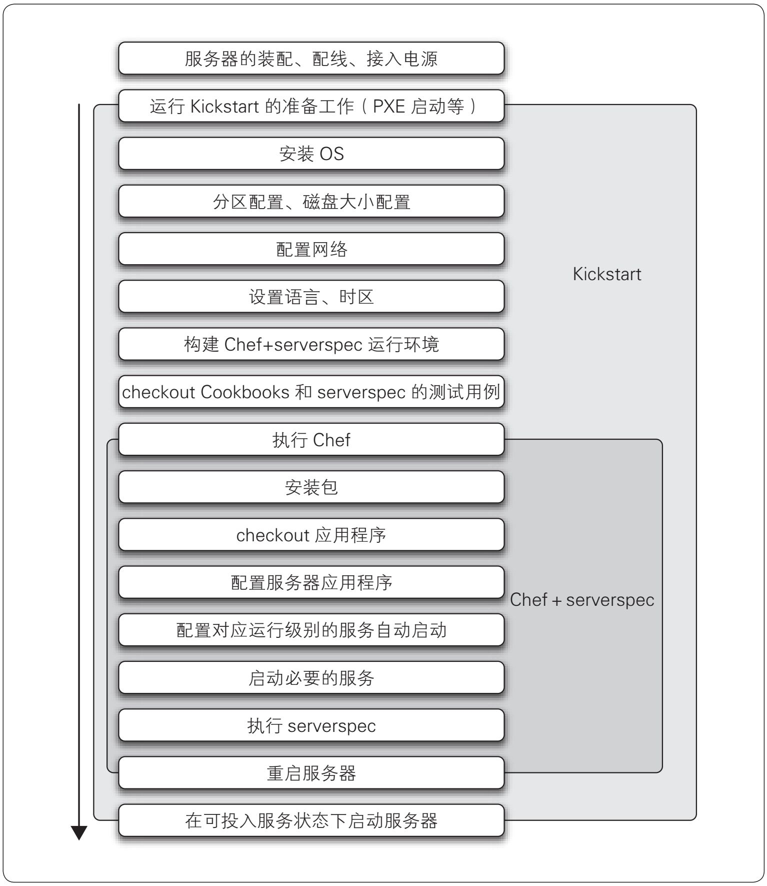
6.4.5 最佳实践（其 2）

因为重视硬件性能或受服务合同上的限制，正式环境中使用物理服

① https://www.docker.io/
② http://puppetlabs.com/
③ http://www.cucumber-chef.org/
④ https://github.com/test-kitchen/test-kitchen

务器[1]的现场不在少数。在这样的情况下，比较理想的是尽量避免手动升级服务器。Kickstart 能够实现物理服务器的 OS 安装的自动化。通过组合使用 Kickstart+Chef+serverspec，能够缩短服务器从投入到开始提供服务的时间（图 6.11）。

图 6.11　通过 Kickstart+Chef+serverspec 实现服务器安装自动化的例子



① 相对于虚拟服务器来说。——译者注

6.4.6　实现物理服务器投入运营为止的所有步骤的自动化

Kickstart 同样适用于虚拟环境的安装，因此可以在 VirtualBox、Xen、KVM 等上面事先对这样的流程进行验证。Chef 和 serverspec 的组合可以用来进行测试，并且通过建立之前叙述的构建服务器的 CI 环境，还可以实现频繁地构建服务器，这样的成功案例已经有很多了，因此可以将执行 Chef 这一步之前的软件包安装以及和硬件相关的分区设置等交由 Kickstart 来处理。

6.5 编配（Orchestration）

6.5.1 发布作业的反面教材

这里的编配大致上可以简单地理解为发布作业的自动化。下面举一些发布作业中的反面教材，如果其中没有一条和你所在的项目相符合的话，可以直接跳过本节。

- 手动进行发布作业
- 发布作业的内容每次都不相同
- 发布作业需要特殊的知识（其他人不知道如何发布）
- 不能反复进行任意次数的发布

什么是发布？发布可以理解为通过 SSH 登录到远程服务器，将代码切换到最新的分支并进行更新数据库等操作。小型项目的发布，经过几个到几十个步骤的操作可能就结束了。而一定规模的项目的发布，则需要经过数百以至于数千个步骤。

手动实施上述工作不仅会耗费大量的时间，对于负责发布的工作人员的体力也是极大消耗。因此这样的情况下要反复地进行发布是不现实的。

对正式环境实施的发布总是会伴随着失败的风险。将发布失败的可能性降为零是不可能的，但有方法让它接近于零。那就是推进发布作业的自动化。在测试环境以及 staging 环境上反复演习自动化的发布并进行验证，在正式环境之前发现问题，然后修改有问题的内容，再反复地对发布进行演练，这样就能降低在正式环境中发生错误的风险。

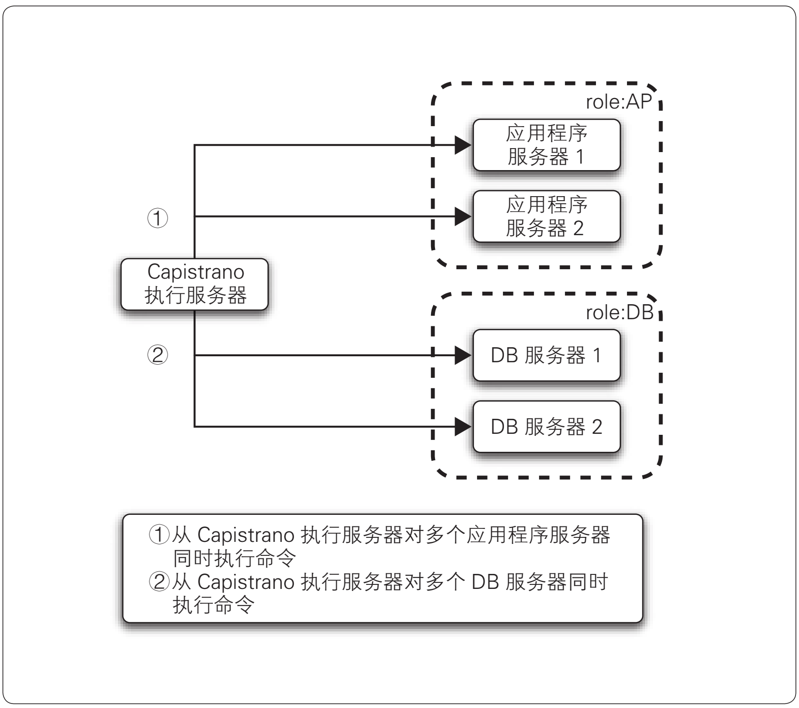
6.5.2 Capistrano

6.4 节中介绍了用 Ruby 编写的 Chef 和 serverspec，发布作业的自动化方面也有用 Ruby 编写的非常强大的工具 Capistrano。这款工具为 Ruby on Rails[①]框架项目的副产品，支持 Ruby on Rails 应用程序的部署。使用 Ruby 以外的语言或框架编写的应用程序的发布同样可以使用 Capistrano，并且已经有了很多这样的实例。

Capistrano 的系统构成

使用 Capistrano 时的系统构成如图 6.12 所示。

图 6.12 Capistrano 的构成



① http://rubyonrails.org/

只需在执行 Capistrano 的服务器上安装即可使用。也就是说，Capistrano 采用的是 Push 型的架构。不需要在发布对象的服务器上安装代理等，只需要确保执行 Capistrano 的服务器能够通过 SSH 登录就可以了。

例如，将多个服务器按角色归纳为应用程序服务器和数据库（DB）服务器，对应用程序服务器执行代码更新的任务，在数据库服务器上为了修改数据库模式而进行 Dump 等任务，这些发布作业都可以通过 Capistrano 一并执行。

像这样简单且频繁的发布作业，在运维中会频繁地出现。频度高的时候每天、每小时都要进行。用 Capistrano 实现上述作业的自动化，无论重复多少次都能够毫无差池地完成。

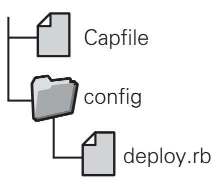
●········ Capistrano 的安装

Capistrano 从安装到初始化只需要执行数条命令即可，因此可谓是使用门槛相当低的工具。

```
$ gem install capistrano
$ capify .
```

执行 capify命令会生成部署配置文件的模板（图 6.13）。

图 6.13　部署配置文件模板的构成



●········ deploy.rb

这里以向 web1 和 web2 这两个应用程序服务器部署文件为例来讲解 deploy.rb 的配置。

```
role :web, 'web1' , 'web2'

task :deploy_task, :roles => :web  do
  upload("/usr/local/src/deploy_file", "/tmp", :via => :scp)
  run "uname -a > /tmp/uname_file"
  download "/tmp/uname_file",
    "/tmp/uname_file.$CAPISTRANO:HOST$"
end
```

使用 upload 向远程的 web1、web2 服务器部署文件，还能够在 web1、web2 服务器上执行通过 run 设置好的命令。本例中会执行 uname命令，并将标准输出定位到 /tmp/uname_file。

最后设置用 download 命令来回收 /tmp/uname_file。保存回收文件的路径中设置有 $CAPISTRANO: HOST$ 这样的变量，这是因为在以多个服务器为对象的情况下，文件名相同会造成原有文件被覆盖，因此用远程服务器名来替换部分文件名。

●……… Capistrano 的执行方法

```
$ cap deploy_task
```

虽然需要事先确保作为执行对象的 web1、web2 服务器能够通过 SSH 登录，但之后只需要记述不满 10 行的配置文件，无需反复操作就能确保执行相同的命令。

即便部署对象的台数增加，也只需在 role 中添加对象服务器，并执行 cap这 1 条命令，这样所有的工作就完成了。从有 1 ～2 台部署的对象服务器时开始，将反复实施的作业做成 task 并由 Capistrano 来实施，这样即便台数增加也应该可以轻松应对了。

6.5.3　Fabric

和 Capistrano 相同，Fabric 也是一款通过 SSH 登录到多台远程服务器，按顺序执行作业的工具，由 Python 编写。功能上和 Capistrano 几乎完全相同，比较大的区别在于 Capistrano 基本上是并行执行处理，而 Fabric 则能够简单地在串行、并行处理间切换。

一般认为为了高效、快速地进行发布作业，并行处理会比较理想，

实际中因为要确保安全地按照手册进行操作，要以串行方式处理的情况并不少见。在 6.6 节中我们将介绍无需停止服务就可以进行的零停机时间部署（Zero Downtime Deployment），如果设定了将两台 Web 服务器依次从网站的负载均衡器上解除，进行更新后再添加回去这样的做法，使用 Fabric 就能够简单地实现。

●……… Fabric（串行执行）的情况

❶ 从负载均衡器解除服务器 A
❷ 更新服务器 A
❸ 将服务器 A 加回负载均衡器
❹ 从负载均衡器解除服务器 B
❺ 更新服务器 B
❻ 将服务器 B 加回负载均衡器

在上述例子中，执行到 ❸～❹ 步骤时，就完成了一半的发布工作，更新完的服务也将公开。

●……… Capistrano（并行执行）的情况

❶ 从负载均衡器解除服务器 A
❷ 从负载均衡器解除服务器 B
❸ 更新服务器 A
❹ 更新服务器 B
❺ 将服务器 A 加回负载均衡器
❻ 将服务器 B 加回负载均衡器

Capistrano 在 ❶～❷ 步时服务停止，和预期的动作不相符。虽然有一些用 Capistrano 实现串行处理的解决方案，但使用直接支持串行处理的 Fabric 更为方便。

●……… 理解本地服务器和远程服务器操作上的区别

首先来看一下 Fabric 基本的记述例子。

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

from fabric.api import *

def fabtest():
    local('mkdir -p /tmp/local_test')
    run('mkdir -p /tmp/remote_test')
    with lcd('/tmp/local_test'):
        local('pwd')    ←lcd/local在本地服务器上执行
    with cd('/tmp/remote_test'):
        run('pwd')    ←cd/run在远程服务器上执行
```

Fabric 能够分开记述本地服务器和远程服务器上的操作。用 local 记述的命令在运行 Fabric 的本地服务器上执行，用 run 记述的命令在远程服务器上执行。在移动当前目录并执行命令的情况下，用 with 语句将 lcd 和 local，以及 cd 和 run 绑定到一起来记述。

下面是获取文件并部署的例子。

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

from fabric.api import *

def getputtest():
    get("/var/log/httpd/access.log")
    put("testfile" , "/tmp")
```

Fabric 能够从本地服务器向远程服务器发送文件，反之也是可行的。

上述两个例子展示了基本的 Fabric 记述方法。下面再来看一下零停机时间发布用的 fabfile.py，发布作业自动化的例子如下所示。

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

import sys, os, re , string
from fabric.api import *

def getenv(name):    ←取得服务器的环境变量
    if os.environ.has_key(name):
        return os.environ[name]
    return ""

def deploy():
    change_balancer('disable')
```

```
    build_update()
    change_balancer('enable')

def change_balancer(mode):
    run('/path/to/change_balancer' + mode)    ←切换负载均衡器的命令

def build_update():
    with cd('/path/to/build_dir/'):
        run('git checkout master && git fetch && git checkout -b %s %s' %
(getenv('RELEASE_BRANCH'), getenv('RELEASE_TAG')))
        run('/path/to/build_dir/application_starter restart')
          ↑重启应用程序的命令
```

Fabric 可以直接使用 Python 的写法，也可以使用 shell 脚本的记述方式，因此无论对于运维人员还是开发人员来说，Fabric 都是容易上手的工具。上述例子中在运行 Fabric 的服务器上将分支名和标签名分别设置给环境变量 RELEASE_BRANCH 和 RELEASE_TAG，在这样的状态下执行 Fabric 就能切换到任意的版本。

●········ Fabric 的运行方法

```
$ fab -H 服务器A,服务器B deploy
```

Fabric 可以在 fabfile.py 中自由地定义 object。这个例子中将 deploy 作为 object 定义并执行。这样就如同“Fabric（串行执行）的情况”中所记述的内容一样，不需要停止服务就可以进行发布作业。上述例子中只涉及了两台服务器，当服务器的台数更多时，如果仍然一台一台地执行的话，发布所花费的时间就会和服务器台数成比例地增加。为了将发布所用的时间控制在一定范围内，可以设置 Fabric 为并行执行，并配置同时执行的数量。

由不少于 4 台服务器运行时的命令如下所示。用 -P指定并行执行，用 -z来设置同时执行的台数。

```
$ fab -P -z 2 服务器A,服务器B,服务器C,服务器D deploy
```

❶ 服务器 A，服务器 B 执行 deploy

❷ 服务器 C，服务器 D 执行 deploy

发布作业可以按照上述顺序进行，如果有 6 台服务器的话，可以将 -z设置为 3，将发布分为前半部分和后半部分。即使进一步增加服务器台数，也可以用和两台服务器时相同的时间完成发布作业。通过有效运用 Fabric，就能够很容易地将我们从手动发布作业中解放出来。

6.5.4 Jenkins

Jenkins[①]作为 CI 工具来说非常实用，作为部署的辅助工具来说同样可以加以有效利用。Jenkins 通过在远程服务器上安装客户端代理（slave agent），可以在远程服务器上执行各类任务。

●……主节点（master node）和从节点（slave node）的协作

Jenkins 客户端代理的安装方法非常简单。确保能够从主节点通过 SSH 登录到从节点，这样只需在 Jenkins 管理画面上添加从节点，主节点就会向从节点部署必要的文件，并自动启动客户端代理（图 6.14）。客户端代理的启动可以根据需要设置为一直启动、定时启动，或只在执行任务时启动等。

图 6.14 从主节点经由从节点执行任务



这里将对象服务器作为节点、将部署相关的 task 作为任务进行管理，通过联系各个事物，来构建自动化发布的环境。和编配工具

① Jenkins 的安装方法请参照 5.4 节。

Capistrano/Fabric 相比，Jenkins 的优势在于具备控制台输出和用户管理功能。

无论是为了能在 Web 的管理画面上看到谁、何时、执行了什么内容这样的记录，还是从留下跟踪信息的观点出发，Jenkins 可以自动实现上述功能。通过添加 Active Directory plugin[①]这款插件，Jenkins 能够和 Active Directory 进行协作。并且通过添加 Role Strategy Plugin[②]插件，Jenkins 可以以任务为单位进行用户权限管理。因此，即便是需要严格进行用户管理的企业，也可以放心使用。

通过利用 Jenkins 的参数化构建（build）这个功能，可以在每次执行任务时传递 shell 变量，以便对任务进行灵活的管理。

下面，我们以从远程服务器上通过 scp命令发送文件为例，对从节点的添加、任务的设置、执行等一系列流程进行讲解。

●········ 从节点的添加

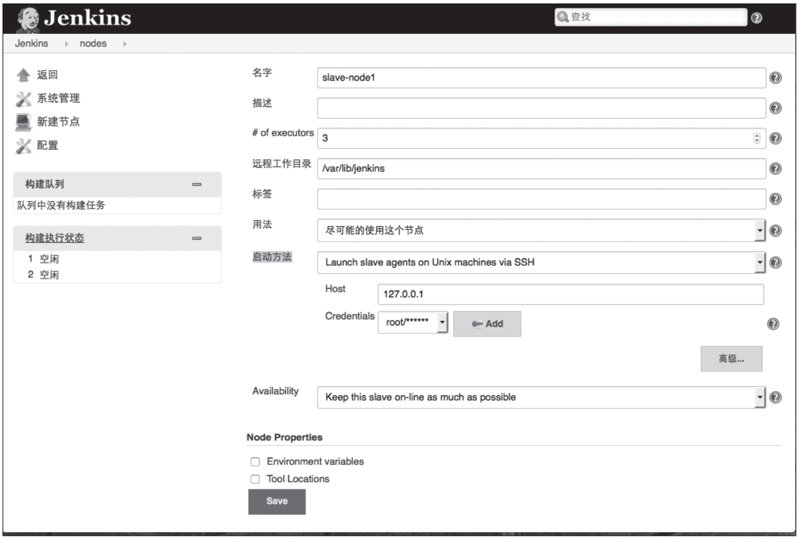
顺利启动 Jenkins 后，通过浏览器访问 http://${remote_host}:8080，能够看到 Jenkins 管理画面的初始状态。该状态允许文件访问，因此正式使用的情况下，建议通过添加用户或防火墙等对 IP 进行适当的限制。

添加从节点时，按照“系统管理”→“管理节点”→“新建节点”这样的流程对节点进行设置。必须设置的项目有 of executors（同时 build 数量）和远程工作目录、启动方法（图 6.15）。

① https://wiki.jenkins-ci.org/display/JENKINS/Active+Directory+plugin

② https://wiki.jenkins-ci.org/display/JENKINS/Role+Strategy+Plugin

图 6.15 节点的添加

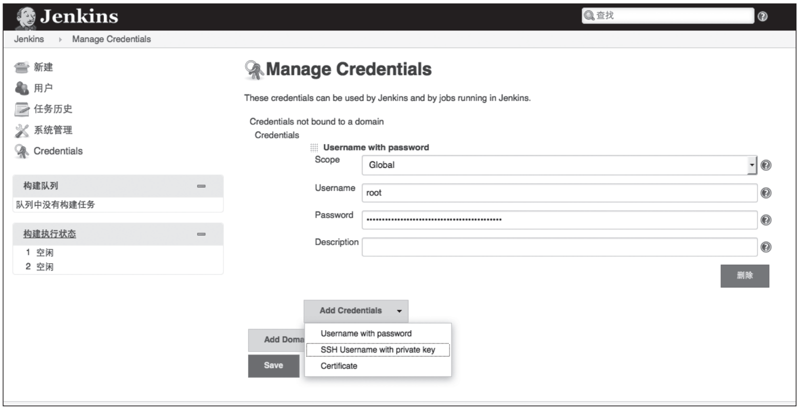


由远程工作目录决定部署客户端代理的路径，如果该目录下还没有部署客户端代理，Jenkins 会自动从主节点向对应的路径进行部署。在启动方法中配置启动客户端代理所需要的路径。

UNIX 系统选择 Launch slave agents on Unix machines via SSH，Windows 机器可以选择 Let Jenkins control this Windows slave as a Windows service。其他启动客户端代理的方法还有 JNLP（Java Network Launching Protocol），该方法适用于主节点在防火墙外，其他的从节点在防火墙内等存在网络限制的情况。

为了通过 SSH 启动客户端代理，需要配置认证信息，这里可以设置 ID/PASS 或密钥（图 6.16）。

图 6.16　认证信息的管理



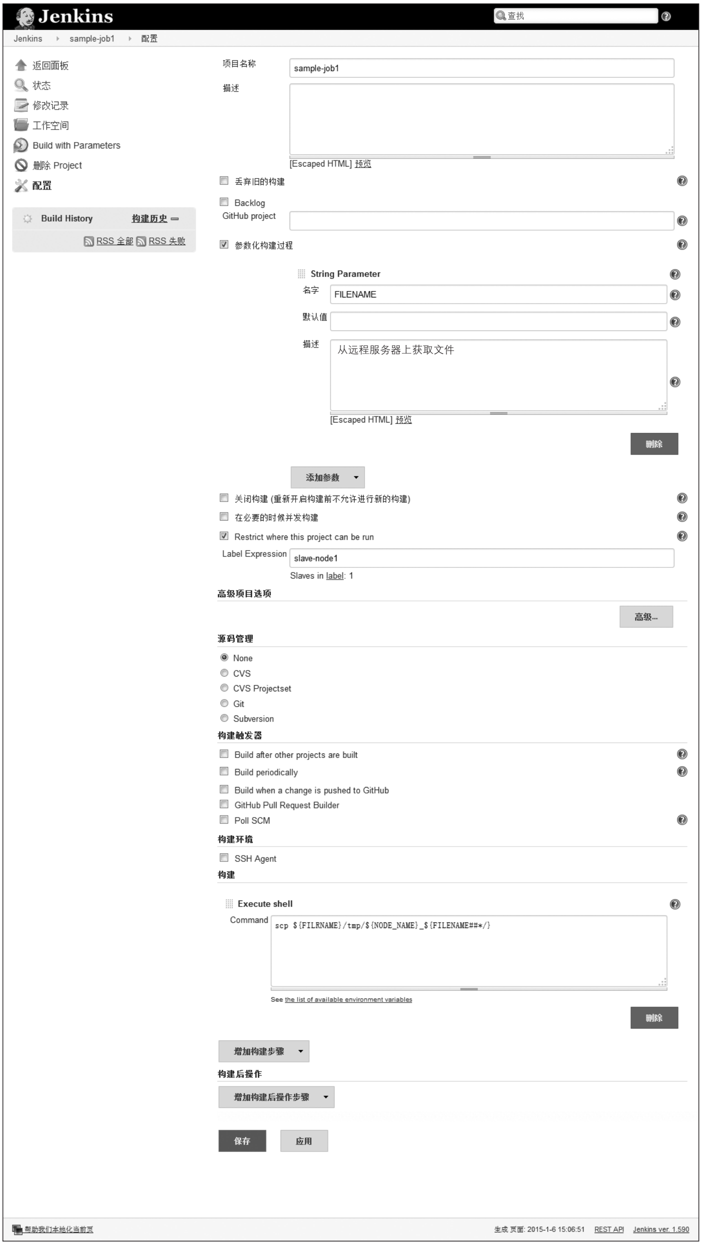
待节点的配置完成并通过认证，远程服务器上会启动名为 slave.jar 的进程，作为主节点的管理对象由主节点控制。如果添加从节点出错，有可能是因为认证信息没有正确设置，或者 bashrc 中记载了多余的命令而造成了 slave.jar 启动失败。

●……… 任务的添加

添加完从节点后，接着添加在从节点上执行的任务。可以从 Jenkins 首页的“新建”开始添加。选择“构建一个自由风格的软件项目”。Jenkins 是作为 CI 工具开发的，所以有构建和代码管理相关的配置，作为编配工具使用时可以直接跳过这部分配置。

在 Restrict where this project can be run 设置刚才添加的节点，点击“构建”下的“增加构建步骤”，选择“Execute shell”，这样就能在远程服务器上执行任意的命令了。作为执行命令时传递变量的方法，可以选中“参数化构建过程”，添加文本、字符串、单选框等输入栏。可以设置多个参数，“名字”一栏中输入的文字能够直接作为环境变量传递给脚本（图 6.17）。

图 6.17　任务的添加


Jenkins
查找
Jenkins
sample-job1
配置
返回面板
状态
修改记录
工作空间
Build with Parameters
删除 Project
配置
Build History
构建历史
RSS 全部
RSS 失败
项目名称
sample-job1
描述
[Escaped HTML] 预览
丢弃旧的构建
Backlog
GitHub project
参数化构建过程
String Parameter
名字
FILENAME
默认值
描述
从远程服务器上获取文件
[Escaped HTML] 预览
删除
添加参数
关闭构建 (重新开启构建前不允许进行新的构建)
在必要的时候并发构建
Restrict where this project can be run
Label Expression
slave-node1
Slaves in label: 1
高级项目选项
高级...
源码管理
None
CVS
CVS Projectset
Git
Subversion
构建触发器
Build after other projects are built
Build periodically
Build when a change is pushed to GitHub
GitHub Pull Request Builder
Poll SCM
构建环境
SSH Agent
构建
Execute shell
Command
scp ${FILRNAME}/tmp/${NODE_NAME}_${FILENAME##*/}
See the list of available environment variables
删除
增加构建步骤
构建后操作
增加构建后操作步骤
保存
应用
帮助我们本地化当前页
生成 页面: 2015-1-6 15:06:51
REST API
Jenkins ver. 1.590

这里定义名为 FILENAME 的字符串变量，shell 脚本中的调用方式如下。

```
scp ${FILENAME} /tmp/${NODE_NAME}_${FILENAME##*/}
```

除了上述参数之外，Jenkins 的任务中还可以使用名为 NODE_NAME 的变量。

●……… 任务的执行

任务添加完成后就是执行了。点击新建任务中的 Build with Parameters，会出现任务中配置参数的输入画面，输入远程服务器上部署文件的路径（图 6.18）。点击开始构建，就可以从远程服务器上获取任意文件。

图 6.18　任务的执行



任务执行的记录会留在“构建历史”中。蓝色的图标表示成功，红色的图标表示失败，一目了然。如果失败的话，点击“构建历史”的链接，通过确认“控制台输出”的 shell 标准输出，就可以确认失败的具体内容。

这个例子中是通过点击构建按键来执行任务的，其他的任务执行时间还可以选择 Build after other projects are built（在其他项目的构建后开始构建）或 Build periodically（定时构建）。Build periodically 能够以 Cron 的形式安排任务执行。因此应用本例中所列举的 scp 任意文件的内容，Build periodically 能够实现从多个从节点上定时收集各类日志这一

作业的自动化。

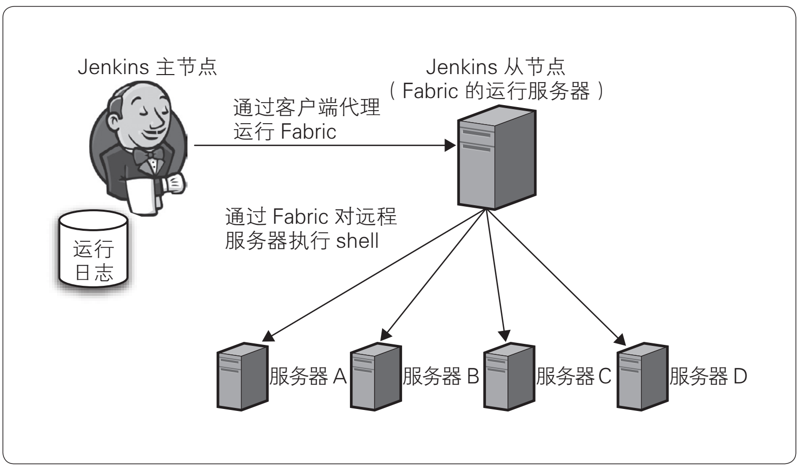
Jenkins 能及时掌握哪个从节点的日志收集处理失败等信息，对失败的处理还可以配置警告，因此可以作为管理工具加以有效利用。

6.5.5 最佳实践

●……… 结合 Jenkins 和 Fabric

至此我们介绍了几款编配相关的工具。并不是说只要选用其中的一个就可以了，只有组合使用上述工具，才能实现先进的自动化环境构建。笔者所实践的自动化部署环境中，用 Jenkins 管理所有的任务，通过 SSH 登录远程主机，执行 shell 的部分则使用 Fabric 来实施（图 6.19）。

图 6.19 Jenkins 和 Fabric 的组合



这个结构的优点在于，通过 Jenkins 的从节点执行 Fabric，能够将 Fabric 的执行结果作为 Jenkins 的控制台日志保存下来。Fabric 添加远程主机非常容易，并且能够作为代码进行管理，因此在版本管理的基础上可以对手头的环境进行验证，或者对各类环境进行操作。

通过组合使用工具，就能够取长补短，提高工作效率。

6.5.6　考虑安全问题

推进发布的自动化需要考虑用 root 账户登录 SSH 并执行命令，或者以有权执行 sudo 的用户进行登录。但从安全方面来说，允许 root 登录是非常危险的，因此很多服务器将 sshd_config 中的 PermitRootLogin 设置为 no。

在这种情况下，可以设置为只有从运行 Capistrano 或 Fabric 的服务器发出的登录请求才允许 PermitRootLogin，禁止除此之外的客户端的 root 登录。

通过设置为只有特定的用户（服务器管理员）才能登录到运行 Capistrano 或 Fabric 的服务器，并采用公钥验证的方式，就能在确保安全性的同时构建自由度较高的发布自动化环境。具体地说就是，对部署目标服务器的 /etc/ssh/sshd_config 进行如下配置。

```
RSAAuthentication yes
PasswordAuthentication no
PermitRootLogin no

Match Address 192.168.1.1/32        ←从OpenSSH4.4开始可以使用Match命令
    PermitRootLogin without-password
    ↑通过设置without-password，允许密码认证以外的登录方式
```

sshd 中可以用 Match 命令对条件分支进行配置，如果是符合条件的 IP 地址或用户发出的 SSH 连接请求的话，可以按照配置的内容进行控制。本例中，假定运行 Capistrano 或 Fabric 服务器的 IP 地址为 192.168.1.1，则仅允许上述服务器对 root 账户发起 RSA 认证的 SSH 连接请求。从安全性方面来说，需要在兼顾各方面的基础上进行有效的设计。

专栏　手动部署的例子

至此我们介绍了几款部署的自动化工具，但实际运用中仍然有不得不手动作业的状况发生。

■无法实现自动化，不得不手动作业的情况

需要手动作业的案例有下面这些。

❶ 第一次构建自动化环境时

❷ 获取修复故障用的特殊日志

❸ 紧急地重启服务

❹ 预料之外的情况导致自动部署停止时，为了下次部署正常运行而进行的修复工作

关于 ❶ ~ ❸，如果今后可能反复发生的话，为了任何时候都能够处理，建议为它们配置自动化。

❹ 的情况下，需要注意的是这里并不推荐在部署失败时强行手动对服务器进行调整。为了通过部署而强行修改配置的话，就会导致验证环境和正式环境产生差异，那么下次部署时就很有可能再次失败。

部署脚本的内容出错的话，应该修正错误并再度实施部署，除此之外的方法这里都不认可。❹ 中的修复工作是指那些在并非由于部署脚本的内容而造成部署停止的情况下进行的修复工作，例如由于磁盘空间不足而造成部署失败时，为了确保剩余空间而删除一些不需要的文件等。

我们不可能提前预测到实际运用中的各种情况并做好准备。1 台服务器的话还可以通过 SSH 登录进行操作，而要对多台服务器实施作业的话，就非常考验耐心和集中力了。

针对这些意外产生的手动作业，也应该尽可能地借助工具，考虑是否可以通过利用工具，实现无论几台服务器都只需要一次输入就能完成作业。

■手动部署的实用工具

表 6.a 中列举了手动作业时的一些实用工具。

表 6.a 手动部署的实用工具

RLogin	Windows 上运行的名为 RLogin[①]的终端能够向所有打开着的连接同时发送按键输入的操作，这样就能对所有登录状态下的服务器同时实施相同的操作
Tera Term	借助常规的终端软件 Tera Term[②]的广播命令功能，可以向多个终端同时发送命令。感觉只操作了 1 台服务器，但实际上能够同时对 10 台、甚至 100 台服务器进行操作，还可以减少操作失误的可能性。需要注意的是，原则上同时向多个目标发送键盘的输入，服务器状态不一致的情况下可能会返回不一样的结果。例如执行 cd/usr/local/tmp/ 命令后再执行 touch test 的情况下，1 台特定机器上的 /usr/local/tmp/ 目录损坏时，test 文件就会被生成在登录时的目录下。因此，在每一步操作之后，应该尽可能地确认下所有的终端，或者设法通过操作来吸收这种环境上的差异
ClusterSSH/Parallelssh/ClusterIt	Linux 或 Mac OS 等 UNIX 系统环境上有 ClusterSSH/Parallelssh/ClusterIt[③] 等的 OSS，能够在多台服务器上同时执行命令

① http://nanno.dip.jp/softlib/man/rlogin/ 并不是使用 513TCP 端口的工具。

② http://www.vector.co.jp/soft/win95/net/se276622.html

③ http://sourceforge.jp/magazine/08/11/06/0025200

6.6 考虑运用相关的问题

结合使用之前介绍过的部署自动化相关的各类工具就能够构建部署流水线。部署流水线建成并进入到维护的循环阶段后，就可以在理想的时间，安全地向正式环境部署新的代码，这样就可以频繁地对正式环境进行部署了。这时需要考虑新的问题，就是该如何处理向正式环境部署过程中发生的服务中断，以及由版本更新所引发的问题。

不解决这些问题就无法对正式环境进行频繁的部署，因此下面将介绍部署时无需中断服务的机制，以及部署后发生的问题的应对方法。

6.6.1 不中断服务的部署方法

多数 Web 应用程序在代码被更新后都需要重启 Web 服务。在请求处理过程中重启，不仅会造成该请求失败，万一启动失败的话还有可能造成服务中断。既然部署作业有可能会造成服务中断，那么在工作日白天进行部署的话就不会被允许吧。无论部署作业的自动化程度有多高，这样也无法实现频繁的部署。因此这里首先介绍不会因部署造成服务中断的机制，即零中断时间部署的相关内容。

6.6.2 蓝绿部署（blue-green deployment）

首先我们来看一下零中断时间部署方法之一的蓝绿部署（图 6.20）。

图 6.20　**蓝绿部署**

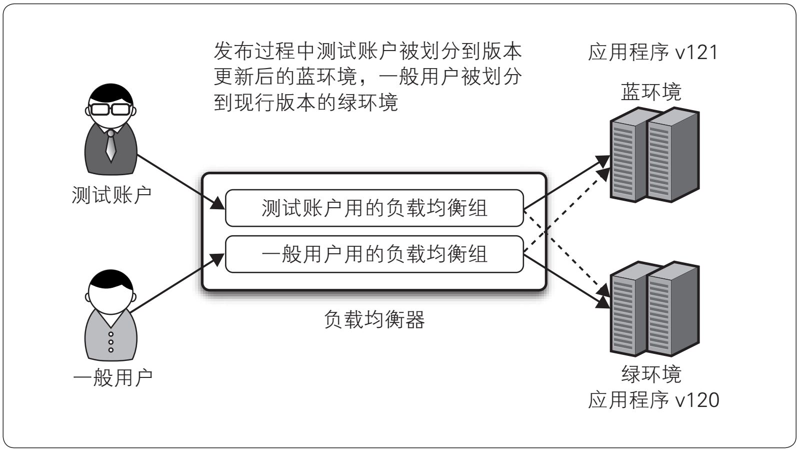


图 6.20 中设置有负载均衡器，后端有两台应用程序服务器分担负载，下面我们就以这样的构造为例来讲解零中断时间部署。应用程序服务器分别分成蓝环境（blue.example.com）和绿环境（green.example.com），负载均衡器采用 Apache 的 mod_proxy_balancer 模块和 mod_rewrite 模块的组合，配置示例如下。

```
#模块的加载以及mod_rewrite的有效化
LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_balancer_module modules/mod_proxy_balancer.so
LoadModule rewrite_module modules/mod_rewrite.so
RewriteEngine On
RewriteLogLevel 1
RewriteLog logs/rewrite.log

#负载均衡器的管理画面用的URL /balancer-manager的配置
<Location /balancer-manager>
    SetHandler balancer-manager
    Order deny,allow
    Deny from all
    Allow from 192.168.0.0/16
</Location>

#测试账户用的负载均衡组
<Proxy balancer://test.example.com>
    BalancerMember http://blue.example.com:8080 loadfactor=50
    BalancerMember http://green.example.com:8080 loadfactor=50
</Proxy>
```

```
#一般用户用的负载均衡组
<Proxy balancer://prod.example.com>
    BalancerMember http://blue.example.com:8080 loadfactor=50
    BalancerMember http://green.example.com:8080 loadfactor=50
</Proxy>


#测试账户的配置
#测试账户的IP地址
RewriteCond %{REMOTE_ADDR} ^10.8.15.1$
RewriteCond %{REQUEST_FILENAME} !^/balancer-manager
RewriteRule ^/(.*) balancer://test.example.com/$1 [P]

#一般用户的配置
RewriteCond %{REQUEST_FILENAME} !^/balancer-manager
RewriteRule ^/(.*) balancer://prod.example.com/$1 [P]
```

在浏览器中输入 `http://${remote_host}/balancer-manager`，访问采用上述配置的负载均衡器，会显示负载均衡器的管理画面（图 6.21）。

图 6.21 负载均衡器的管理画面

Load Balancer Manager for loadbalancer.example.com

Server Version: Apache/2.2.15 (Unix) DAV/2
Server Built: Feb 13 2012 22:31:42

LoadBalancer Status for balancer://test.example.com

StickySession	Timeout	FailoverAttempts	Method
-	0	1	byrequests

Worker URL	Route	RouteRedir	Factor	Set	Status	Elected	To	From
http://blue.example.com:8080			50	0	Ok	0	0	0
http://green.example.com:8080			50	0	Ok	0	0	0

LoadBalancer Status for balancer://prod.example.com

StickySession	Timeout	FailoverAttempts	Method
-	0	1	byrequests

Worker URL	Route	RouteRedir	Factor	Set	Status	Elected	To	From
http://blue.example.com:8080			50	0	Ok	1	0	0
http://green.example.com:8080			50	0	Ok	1	0	0

Apache/2.2.15 (CentOS) Server at loadbalancer.example.com Port 80

如图 6.21 所示，可以对发送至测试账户的负载均衡组和一般用户的负载均衡组的蓝绿环境的请求进行配置。在上述环境下，实现蓝绿部署的步骤如下所示。

❶ 将 balancer://test.example.com 的 green.example.com 设置为 Disable
❷ 将 balancer://prod.example.com 的 blue.example.com 设置为 Disable
❸ 对 blue.example.com 实施自动部署，更新应用程序
❹ 从 IP 地址为 10.8.15.1 的机器执行测试（被划分到测试用的负载均衡组），到下一个步骤为止的时间段中，一般用户的请求被分配到 prod.example.com，服务持续提供
❺ 通过测试后，将 balancer://test.example.com 的 green.example.com 设置为 Enable
❻ 将 balancer://test.example.com 的 blue.example.com 设置为 Disable
❼ 将 balancer://prod.example.com 的 blue.example.com 设置为 Enable
❽ 将 balancer://prod.example.com 的 green.example.com 设置为 Disable。这时一般用户可以开始使用新版本的服务
❾ 对 green.example.com 实施自动部署，更新应用程序
❿ 从 IP 地址为 10.8.15.1 的机器上执行测试
⓫ 通过测试后所有的负载均衡组都设置为 Enable，部署作业结束

以这样的流程进行部署，应用程序更新过程中就无需停止服务。mod_proxy_balancer 的各个负载均衡组的 Disable 和 Enable 的切换可以通过 HTTP 来操作，因此可以使用 Mechanize 这样的程序来控制。蓝绿部署时需要注意的是，通常不进行发布时，为了不浪费资源配置，会将一般用户分别均分到蓝环境和绿环境，而发布作业时由于需要降级运行[①]，因此要选择用户较少的时间段，或者平时保留一定接入能力的冗余。

6.6.3　云（cloud）时代的蓝绿部署

AWS 等 IaaS（Infrastructure as a Service）能够瞬间添加服务器资

① 降级运行是指在部分停止系统的功能或性能的状态下维持系统运行。——译者注

源，这个特长同样能够运用到零中断时间发布上。以刚才的蓝绿部署的应用为例，可以采用这样的方法：平时只运行绿环境这一套系统，仅在发布时利用 IaaS 功能制作绿环境的镜像，再从镜像启动蓝环境进行发布作业。只要能实现上述模式的发布，就基本上可以克服一般的蓝绿部署的弱点了。

发布时无需降级运行，平时也不需要准备双份的服务器资源，就能够随时实施发布。缺点在于需要增加镜像的制作、服务器的启动，以及发布结束后删除蓝环境这一系列的工作。发布作业变得复杂，失败的风险也随之上升。

在服务器启动时采取下列对策能有效地降低风险，可以在对比这些对策的准备工作所需的开销以及所带来的收益的基础上再进行探讨。

- 服务器启动后用 serverspec 实施自动测试
- （AWS 的情况下）服务器的启动操作用 awscli 等工具来实现自动化
- 平时在 staging 环境上对一系列部署的自动化进行测试

6.6.4 回滚（rollback）相关问题的考察

随时准备好退路

无论事前进行多么充分的测试，对正式环境实施发布总是隐藏着发生意外的危险。比如，用户使用了超出 QA（品质保证）部门预测的服务等，进而造成正式环境发生故障。

也就是说，要时常意识到发布时有可能将 bug 也一同发布出去。需要根据 bug 的严重程度，判断是进行回滚还是在下次发布时修正。因此在进行发布之前，要准备好随时回滚的机制。反过来说就是，不应该实施无法回滚的发布作业。

数据库模式的版本管理

代码的回滚只需要回退到升级前运行的版本，多数情况下这样就可

以了。一般的 Web 应用程序将持久性数据保存在 RDBMS 中。如果需要回滚数据库的话，可以使用数据库迁移工具[①]。

●……… 回滚的验证

假设有下面这样的作业流程，我们需要验证在哪个阶段失败，以及如何回滚。

❶ 显示维护画面
❷ 停止应用程序
❸ 更新源代码
❹ 更新配置文件
❺ 更新数据库模式
❻ 启动应用程序
❼ 实施冒烟测试
❽ 实施性能测试
❾ 解除维护画面
❿ 报告发布结束

最理想的情况是能够为所有阶段的失败提供自动化的回滚机制，但这样会耗费高昂的成本。不过至少也一定要准备好确实可行的回滚机制，即使是手动的也行。手动回滚的话，Rlogin 等辅助工具是非常有用的。

下面我们来考虑一下在两个比较主要的回滚点发生问题时的处理流程。

●……… 只更新代码的发布时的回滚

例如在步骤 ❸ 发布作业中断，无法继续进行发布的情况下，回滚的步骤如下。

❶ 将更新了的代码回退到以前的版本
❷ 启动应用程序

① 这款工具在第 3 章中有详细介绍。

❸ 实施冒烟测试

❹ 解除维护画面

❺ 报告发布延期

这是最简单的流程，只需退回到之前的版本就可以了。但是回滚结束后必须用 serverspec 等对服务器进行测试，确认程序是否正常回退，并实施冒烟测试，确认应用程序是否正常运行。

●……… 数据库模式更新时的回滚

到冒烟测试为止都正常运行，在步骤 ❽ 时发现比起发布前，一部分功能的性能慢了 10 倍，这样的情况下该怎么做呢？当然，发布前的各种测试必须对这类问题进行覆盖，但用测试用例覆盖所有问题是不可能的。长期运营的系统会经常遇到这类问题。

例如，特定用户的特定项目的数据量极端大的情况下，或者是性质或倾向不确定的多用户系统中，随着各类使用模式的增加，性能有时会发生退化。发生性能相关问题的情况下必须立即进行回滚。

如果没有上述步骤 ❺ 的话，如上所述，只需回滚代码就能解决问题。即便是能够立即修复的问题也请先行回滚，经过通常的流程后再次进行发布。

由步骤 ❺ 造成性能相关的问题时，以下情况可以通过回退到原来的状态，也就是说通过回滚数据库模式来解决。

- 系统允许该数据丢失（为了收集针对新功能的日志而存储的数据等）
- 更新的内容是数据迁移或规格化、非规格化等不会造成数据丢失的作业

但是也存在下面这些无论如何都无法回滚的情况。

- 系统不允许该数据丢失
- 回滚时有可能产生数据一致性问题

这样的情况下无法回滚数据库模式。因此要预先确认更新的数据库

模式属于上述哪种形式。

可以回滚的模式没有什么需要特别担心的，只要使用迁移工具实施预先测试过的回滚操作，修正 bug 后重新进行发布即可。

无法回滚的模式虽说并非完全没有规避的方法，但为此需要建立大规模的机制，非常麻烦。对于这样的高风险作业，建议设置停机维护时间，在对用户影响较小的日期、时间实施发布。具体做法这里就不再详述了。万一发生问题的情况下，将风险最小化，并尽早解决问题，这样的方案是比较现实的，并且成本方面也可以说是最佳的。

另外，有 1 种在数据库模式更新后即便产生 bug 也可以不回滚数据库模式的技术，那就是将现行的代码和下次发布的代码都运行在数据库更新后的状态下。

测试时请确认即使只更新数据库的模式，现在的代码也可以通过测试。

只要确保即使先只更新数据库，程序也能够正常运行，就不必回滚数据库模式了。因此在步骤 ❽ 发现比发布前性能慢 10 倍的情况下，只需要回退代码即可。

采用这样的机制必须能够分别实施代码的版本管理和基于迁移工具的数据库的版本管理，并且可以使用 Fabric 或 Capistrano 来实现各个部署、回滚的自动化。

6.7 本章总结

至今为止全手动部署的环境想要实现全部自动化，应该从哪里着手？回答是“先去寻找协助者和出资方”。

部署的自动化是从开发人员、QA 团队到运维团队各部门通力协作的产物。请耐心地向他们讲解推进部署自动化会带来哪些好处，以推进团队齐心协力。

除了团队的齐心协力之外，新的环境以及构建新环境所需的时间也是必不可少的。要向出资方、公司的管理层讲解部署自动化的相关内容。如果你所在的公司拥有充沛的资源，并且提供了类似于 Google 的“20% 规则”这样的研究时间，可以在讲解前先着手进行自动化所需的工作，通过演示可运行的系统来进行讲解。如果能够频繁地向客户提供价值，对于大多数公司来说都会产生好的影响。

如果团队内部有不合作的成员，最初他们可能会觉得麻烦而不遵守既定的规则，即便这样也请耐心地推进自动化。当部署自动化开始逐渐渗透到整个团队、整个公司时，它就成为必不可少的存在了。

原来的部署需要专门的知识和经验，如果部署的自动化开始正常运转，那么这样的知识及经验就不再需要了。工作人员也可以通过维护自动化的环境，夜间得以安睡，节假日可以享受私人时间。而从公司的角度来说，对于能够让工程师幸福的部署自动化，也没有理由不实施。

专栏　PaaS 的使用方式

什么是 PaaS

这里并不是要否定本章的内容，但构建自动部署的环境总要花费相当的成本。当然通过实现自动化应该能够获得与其投资相符合的收益，但那些因为项目生命周期短而希望投资最小化的情况下，或者是创新公司希望尽早获得用户反馈的情况下，可以考虑使用

PaaS（Platform as a Service）。

PaaS 是一种由运营商负责提供从网络、OS 到应用服务器等中间件的服务。因为 PaaS 能够立即提供可使用的应用程序运行环境，所以开发人员能够专心于开发工作。

PaaS 在国内外有 Heroku、Force.com、Google App Engine、Cloud Foundry、OpenShift、AWS Elastic Beanstalk、DotCloud、Windows Azure、IIJ GIOMOGOK、NIFTY Cloud C4SA 等众多提供商。

选择 PaaS 时要注意可使用的语言及语言的版本、应用服务器（Apache Tomcat/jetty 等）、可使用的数据库服务、API 的提供、可扩展性、使用成本、插件的种类等。

以 Heroku 为例，登录账户后从管理画面上传应用程序，只要向指定的远程目录实施 git push，就能启动 Web 服务了。无需安装或调整 Apache，只需在管理画面增加名为 Dyno 的 Heroku 特有的 Web 服务的进程数量，就能够处理大量的请求。开发人员只需要 10 分钟就能够搭建起可扩展的 Web 服务并公开。

PaaS 的适用场景

PaaS 能够方便地应用于初期的小规模开发，还能够调整规模。针对这样的特点，这里介绍几种有效的使用方法。

- **创新公司希望尽早公开服务**

  没有足够的资源构建开发环境、测试环境、正式环境的情况下，不要直接仅构建正式环境并公开服务，而是利用 PaaS 获取用户的反馈并反映到产品中，还可以筹措资金等。

- **无法预测峰值负荷的服务**

  提供手机游戏等峰值无法预测的服务时，可以利用易于扩展规模的 PaaS。当服务步入正轨，形成可预测负荷的运营体制后，再来考虑转移到公司内部运维等选项。

- **生命周期短的服务**

  例如限定期限为 1 个月的展会网站等，也可以使用 PaaS。展会开始时收到大量请求的情况下，可以预先扩展规模进行应对。展会结束后再缩减规模，这样就能够以最少的成本维持服务运营，客户解约的话也可以立即停止服务。像

这样，无需负担多余的资产，非常适合于短时间的使用。

- **融入到企业内部的部署流水线中**

  很多 PaaS 都可以使用全球公开的 URL，因此可用于营业用的环境，或者为了获取特定顾客反馈的 demo 环境，这是个很大的优点。

有了 PaaS 就不需要部署自动化了？

在适合利用 PaaS 的场景范围内，也许可以说不需要环境的配置。但是，PaaS 也并不是万能的。使用 PaaS 之前，请仔细考虑 PaaS 的下列缺点。

- **服务的使用量超过一定限度后，费用会激增**
- **vendor lock-in[①] 的风险**
- **PaaS 的限制和应用程序的特性不相符的情况**
- **发生问题时，有时难以获得调查或分析原因用的日志**
- **向云运营方提出的监查要求可能不被接受**
- **想要的服务等级（service level）和合同内容或 SLA[②]（Service Level Agreement）不相符的情况**
- **存在因无法确定问题原因而难以追究责任的情况**

当服务的规模增长到一定程度，PaaS 的费用和提供的服务等级开始变得不相符，或者由于公司的规则原本就和 PaaS 的限制不相符合等情况下，就很难继续使用 PaaS 了。在规划新的服务时，建议综合考虑应用程序的特性，探讨将程序部署到哪里。并且结合定期发布后服务的增长，需要重新考虑怎样的环境才是合适的。

离开 PaaS 的时机

可以考虑将在 PaaS 上公开的应用程序移植到 AWS（Amazon Web Services）等的 IaaS 环境，或者自行运营的公司内部环境中。例如，由于超过了预计的访问量导致 PaaS 的使用成本变得无法接受等

① 过度依赖特定供应商的独特技术进而无法替换为其他供应商提供的同类产品。——译者注

② 服务等级相关的事前协议。主要是系统正常运行率相关的赔偿等内容。

之前叙述的缺点超过使用 PaaS 的优点时，就有必要移植应用程序。

那么，移植时必须注意哪些事项呢？主要有以下 3 点。

- **应用程序运行环境的构建**
- **数据的移植**
- **服务的切换**

第 1 点就是要构建新的应用程序运行环境。没有托管的服务器组或者自己的数据中心的情况下，可以使用 AWS 等的 IaaS 环境。构建环境时可以一边参考本章内容一边进行。

第 2 点是处理持久性数据时，需要对数据进行移植。如果持久性数据保存在 RDBMS 中，可以取得 Dump 数据（导出数据），并预先确认好导入数据的操作步骤。导入、导出所耗费的时间和切换环境时服务中断的时间直接相关，因此需要事先验证需要消耗多少时间。支持增量备份的话，可以预先备份好实施移植之前的部分，这样能够使服务中断的时间达到最小。

第 3 点是服务的切换。如果使用域名的话，可以预先缩短 DNS 记录的 TTL（Time To Live，报文的有效时间），为能够在瞬间完成切换做好准备。切换时禁止访问旧的环境（PaaS），可以通过显示维护画面来防止再向旧的环境存储数据。无法沿用之前的 URL 的情况下，需要考虑访问旧环境时重定向到新环境的调整方式。移植期间需要保留旧的环境，因此可以在确认访问量的基础上来决定关闭旧环境的时间。

第7章

回归测试

7.1 回归测试

至此，我们在第 5 章讲了持续集成（Continuous Integration，CI），在第 6 章讲了持续交付（Continuous Delivery，CD）的相关内容。本章来讲解一下持续测试的相关内容。

7.1.1　什么是回归测试

为了使 CI、CD 更为实用，回归测试就变得必不可少。回归测试是以检查退化为目的的测试。虽然在第 5 章接触了自动化测试，但那只是从开发人员视角出发的单元测试。本章将更进一步，来看一下从用户视角出发的回归测试，即集成测试、用户验收测试的相关内容。

一般来说这类测试和使用系统的用户一样，利用浏览器进行输入或页面迁移，并测试是否得到所期待的结果或页面。因此执行一个个测试用例的时间往往变得很长，如果是长年运维的系统，到所有测试执行结束为止耗费大量时间的情况并不罕见。

事实上，在笔者所在的公司，为了确认前一天提交的结果是否通过回归测试，往往要耗费数天的时间。为了解决这个问题就需要并行执行测试。

本章将讲解为了实现针对 Web 服务的回归测试自动化所必需的并行测试的相关工作，以及具有代表性的用户验收测试自动化工具 Selenium 的使用方法和一些小技巧。

并且，以确保产品质量为主要职责的工程师，往往并不擅长编程。这样的情况下应该如何实现高效的自动化回归测试呢？对于这个问题笔者也将给出自己的答案。

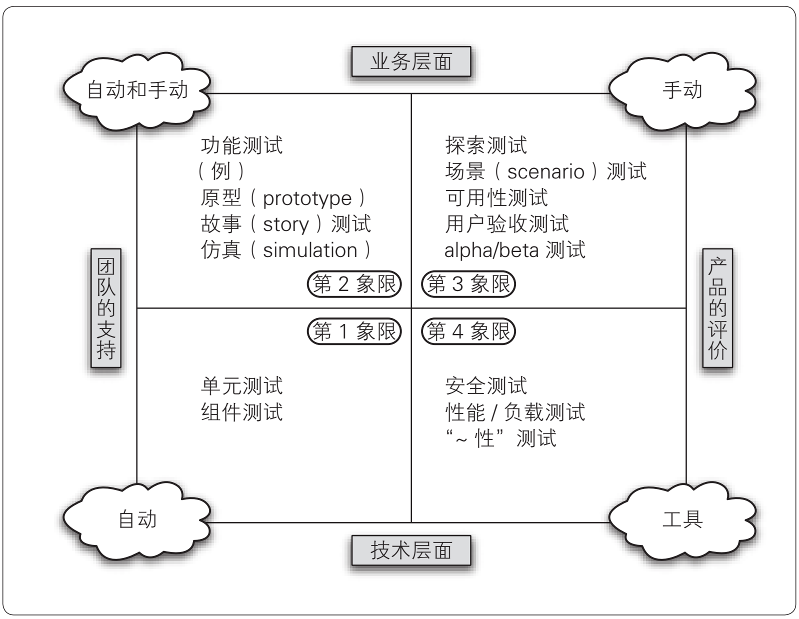
7.1.2 测试分类的整理

产品开发中需要根据不同目的实施各种种类的测试，并且其数量之多，往往会将人压得喘不过气来。因此在进入正题之前，首先对测试这一概念进行简单的整理。开发流程中应该如何计划并实施测试，以及什么种类的测试和发现怎样的 bug 有关，了解上述内容是进行测试的第一步。

测试可以被分为 4 个象限[①]（图 7.1）。这个分类方法将各种测试依据“业务层面看到的”和“技术层面看到的”，以及“以支持团队为目的进行的”和“以评价产品为目的进行的”的不同，用 2 根轴分为 4 个象限，并且在 4 个方块中给出了各类测试的主要手段。

下面对各象限的概要进行讲解。

图 7.1　测试的 4 象限



① （美）Janet Gregory、Lisa Crispin 著，孙伟峰、崔康译《敏捷软件测试：测试人员与敏捷团队的实践指南》，清华大学出版社，2010 年。

●……… 支持团队的技术层面的测试（第 1 象限）

基于单元测试的测试驱动开发（Test Driven Development，TDD）等开发中的主要测试。主要是和内部品质相关的内容，代码的版本管理系统、集成开发环境（Integrated Development Environment，简称 IDE），以及编译自动化工具等也属于这个范畴。

●……… 支持团队的业务层面的测试（第 2 象限）

与第 1 象限偏技术相对，该象限是更接近于业务的顾客观点的测试。除了功能测试以外，原型、仿真、需求以及画面设计等以推进开发为目的的活动也可以作为一种测试在此进行。

并且，这里的测试是用负责业务的工作人员能够理解的语言或格式来记述的，这些测试中的大多数都应该、且已经实现了自动化。该领域还包括借助 Fit[①]、Selenium、Cucumber[②]、Watir[③]以及 Canoo WebTest[④]等工具的自动化测试。

●……… 评价产品的业务层面的测试（第 3 象限）

从业务的视角出发，为了评价产品而进行的测试。探索测试[⑤]、可用性测试[⑥]以及用户验收测试[⑦]属于这个象限。

基本上是以对用户接触的部分进行测试为中心，相对于第 1、第 2 象限侧重于自动化方面，这是只能由人手动实施的测试。

---

① http://fit.c2.com/

② http://cukes.info/

③ http://watir.com/

④ http://webtest.canoo.com/

⑤ 一种探索着进行的测试。不是根据事前准备好的测试计划来执行测试用例，而是一边熟悉产品一边对可疑之处进行测试，并根据测试结果随机应变地进行下一步测试。

⑥ 从用户的视角来确认产品的 UI（用户接口）哪里有问题的测试。

⑦ 检验完成后的产品是否满足需求规约的测试。

●……… 使用技术层面测试的产品评价（第 4 象限）

性能测试、安全性测试等，基本上不使用技术就难以实施的测试。

◆◆◆

针对开发流程中应该注重哪些目的，或者说可以省略哪一部分，可以对照着测试的 4 个象限来确立测试方针。另外，为了确认正在实施或者即将实施的测试是出于什么目的的等，也可以参考测试的 4 个象限，这样在团队以及组织内容易达成共识。

7.1.3 回归测试的必要性

●……… 退化（degrade）的发生

由于系统的版本更新，在之前的版本中正常运行的功能变得无法正常运行，或者紧急修正了某个问题，但引发了其他新的问题……想必谁都经历过这种退化的现象吧。

虽说如此，为了在代码变更时保证质量，既然无法预测系统的何处会发生退化，就必须实施回归测试以确认没有预想外的影响发生。

并且，如果通过回归测试发现了 bug，为了确认 bug 还需要更进一步实施回归测试。但是这样的测试流程需要较多的时间，因此如果不实现自动化，在实际项目中就很少会被运用。

●……… 应该实现自动测试的原因

那么应该怎么做呢？从修改的代码确定影响范围，再从重要程度等风险的观点出发确定测试范围后实施测试，这可能是一般的作法。但是笔者认为通过实现回归测试的自动化，每次都执行所有的测试用例，能够最有效地保证质量。

人为判断执行哪些测试用例本身就可能产生失误。根据时间的限制来决定可以执行的测试范围和测试量，最终只要不执行所有的测试用例，还是难免会担心。

实现测试的自动化还会带来其他的一些好处。正如我们在第 5 章中

说明的那样，每次提交后执行测试，如果能够即刻发现问题，那么问题就能被迅速地解决。

若提交 1 周后才收到当时的修改引发了新的问题这样的报告，有时就需要先回想一番后才能着手。因此实现测试的自动化，频繁地实施测试还能够加快开发速度。

为了减少退化 bug 要怎么做才好呢？在此之前人们为了减少退化 bug 进行了各类尝试，得到的答案是：通过人工测试来保证质量，从效率以及速度上来说都有其局限性，只有通过自动测试才行。从在此之前的手动测试转变为重视自动测试，发生退化的数量也得以大幅减少（图 7.2、图 7.3）。

图 7.2 通过自动测试发现的 bug 在所以被发现的 bug 中所占的比例

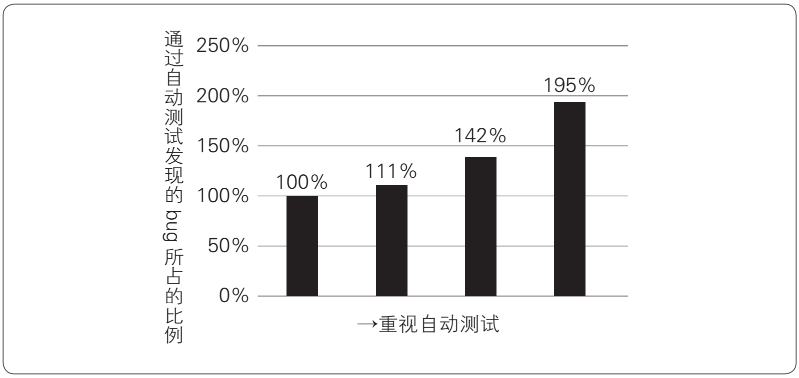
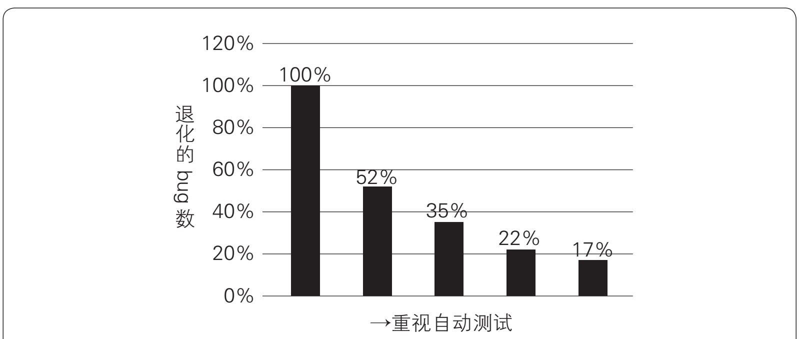


图 7.3 退化数量的变化



当然自动测试的效果根据产品的特性会有所变化，但不难看出，无论是对于提高产品的质量还是顺利推进开发来说，回归测试的自动化都是重要的因素。

7.1.4 回归测试自动化的目标

说起实现回归测试的自动化，之前叙述的各个种类的测试中哪些应该实现自动化呢？首先，一般来说是由开发人员实施的单元测试。可是无论单元测试的覆盖率如何提升，仅靠单元测试是无法保证产品的质量的。单元测试作为确保程序的动作和开发人员的意图相一致的测试，即使达到 100% 的覆盖率，还是有如下这些 bug 无法发现。

- 在程序中的 for 或 while 这样的循环中，不进入循环体或者超过了最大循环次数等边界值相关的 bug
- 包含预料之外的数据时的异常处理不充分而导致的 bug
- 共享内存的操作或数据库的锁处理等，由于进程、线程间共享数据而在特殊的时间点发生的 bug
- 使用了中间件的某个特定版本（Web 服务器、数据库服务器、邮件服务器等）而产生的 bug
- 需求自身的问题所造成的 bug
- 功能未完成而造成的 bug

多数系统会使用数据库服务器或缓存服务器这样的中间件，或者通过 API 调用其他 Web 服务的资源，很多功能仅有系统自身无法运行，因此就需要在和实际产品相同的环境下进行测试。并且还有可能发生和用户所使用的 OS 及浏览器相关的 bug。因而需要对各种条件下系统的运行要求是否得到满足进行测试。

所以，不仅仅是单元测试，还需要从最终使用系统的用户视角出发的集成测试。不过值得庆幸的是，这种集成测试的自动化工具正在不断地被开发出来。下面讲解的作为 Web 应用程序测试框架的 Selenium 就是一款著名的测试工具。

7.2 Selenium

7.2.1 什么是 Selenium

Selenium①是实现 Web 应用程序的功能测试以及集成测试自动化的浏览器驱动测试工具群，具备以 Web 应用程序的测试自动化为特点的各类功能。

和使用系统的用户相同，在浏览器上进行的鼠标操作、在表单中输入文字或者打开页面时，Selenium 能够检查页面的内容是否正确显示、检查表单的值以及验证页面内执行的 JavaScript 等。因此借助 Selenium 能够重复实施至今为止手动操作的测试。

7.2.2 Selenium 的优点

下面介绍一下 Selenium 的主要特点。

●……… 自动化测试用例制作简单

首先是测试用例的制作非常简单。Selenium 提供了名为 Selenium IDE②的工具。该工具具备记录（capture）键盘或鼠标的操作并播放（replay）的“capture・replay”功能，能够非常简单地制作测试用例。

因为能直接根据浏览器上的操作生成测试用例，所以不善于编程的工程师也能简单地制作自动测试用例。

●……… 支持多种浏览器及 OS

其次是测试可使用的浏览器种类多样。现在支持的浏览器有 Internet

① http://www.seleniumhq.org/

② 还有名为 Selenium Builder 的 Selenium IDE 的后续工具。

Explorer（简称 IE）、FireFox、Chrome、Safari、Opera、iPhone 标准的浏览器（Safari）、Android 标准的浏览器，可以说是支持了所有主要的浏览器。并且还支持 Windows、OS X、Linux、Solaris 等多种 OS。

无论对功能进行如何详尽的测试，如果在 FireFox 上正常运行，在 IE 上无法运行的话，这样是没有意义的。因此 Selenium 支持在各种浏览器上执行同样的测试用例。

在系统支持的所有浏览器上手动执行同样的测试是一件非常无聊且麻烦的工作。这样繁琐的作业就应该实现自动化。

7.2.3 Selenium 的组件

如上所述，Selenium 作为自动测试 Web 应用程序的工具群，是由多个组件（工具）构成的。根据运行方式的不同，所使用的组件也有所差异。这里简单地介绍一下具有代表性的 Selenium IDE、Selenium Remote Control 和 Selenium WebDriver。

●……… Selenium IDE

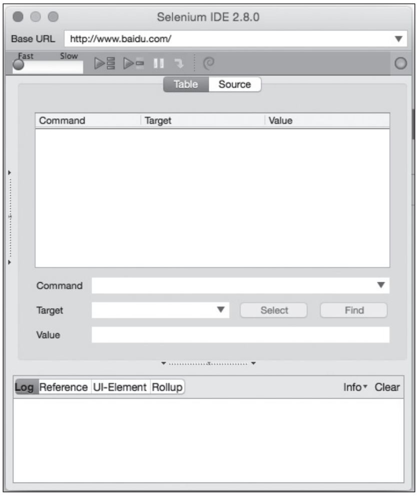
Selenium IDE[①]是 Selenium 测试用的 IDE，是一款支持从测试用例的制作到执行、调试的 Firefox 的插件（图 7.4）。

Selenium IDE 能够通过录制浏览器的操作来制作测试用例，编辑录制的测试用例时还支持自动补足（autocomplete），因此非常方便。并且制作的测试用例能够当场立即执行，还能够通过在测试用例中设置断点来确认执行暂停时的状态，可以说是高效地制作测试用例必不可少的工具。

没有使用过 Selenium 的人，可以从使用 Selenium IDE 开始，来熟悉 Selenium 的测试用例。

① http://www.seleniumhq.org/projects/ide/

图 7.4　Selenium IDE



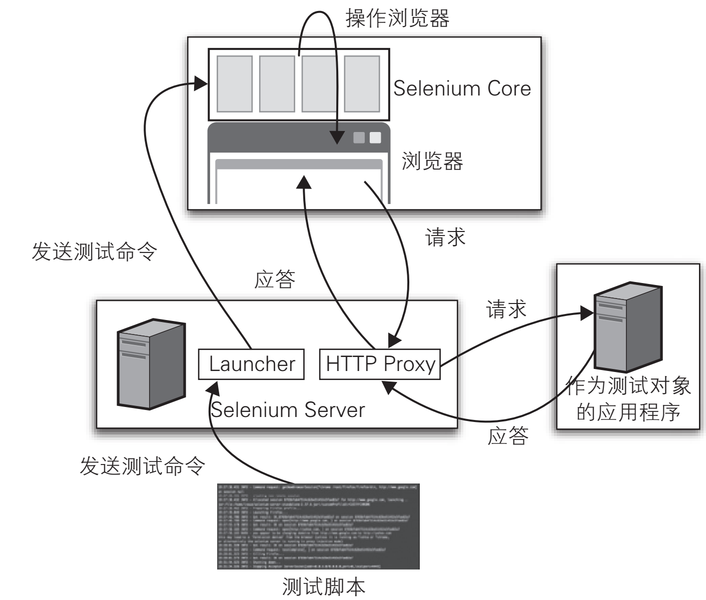
●········ Selenium Remote Control（Selenium RC）

Selenium Remote Control[①]（以下简称 Selenium RC 或 Selenium1）和 Selenium IDE 一样也是自动测试 Web 应用程序的工具（图 7.5）。但是，不同于 Selenium IDE 自身就能制作测试用例并执行测试，Selenium RC 需要在运行浏览器的机器上启动名为 Selenium Server[②]的服务器，该服务器用于中继测试脚本（测试用例）和浏览器，并通过该服务器来执行测试。

① http://www.seleniumhq.org/projects/remote-control/

② Selenium Server 就是 Selenium WebDriver 发布之前的称为“Selenium RC Server”的中继服务器，和 WebDriver 合并成为 Selenium2 后开始改名为“Selenium Server”。

图 7.5　使用 Selenium RC 的构造



Selenium RC 可以启动、终止测试用的浏览器、执行测试脚本上的浏览器操作命令等，还能够起到类似于 HTTP 代理的功能，因此可以在 Firefox 以外的多种浏览器上执行测试。并且 Selenium RC 还提供了各种语言[①]能够调用的 API，因此可以使用各种编程语言制作测试用例。利用编程语言来制作测试用例，可以实现循环处理以及分支条件，还可以将共通处理做成库，因此可以制作出更为高效的测试用例[②]。

●……… Selenium WebDriver

Selenium 是一款非常优秀的测试工具，但由于操作浏览器的引擎部

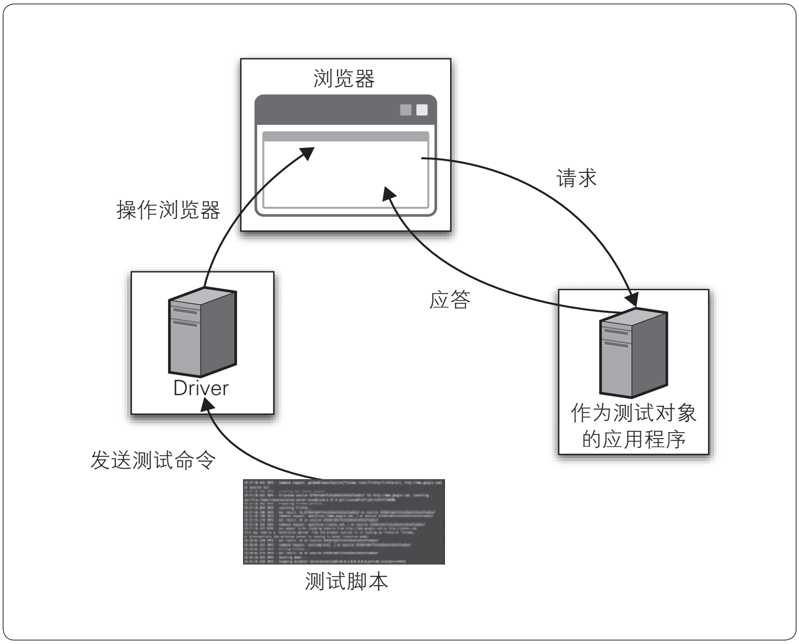
① 主要可使用的语言有 Java、Ruby、Python、Perl、PHP、.NET。

② 即将发布的 Selenium3 将不推荐使用 Selenium RC。虽然本书提到了较多关于 Selenium RC 的内容，但开始新的项目时还是推荐使用 Selenium WebDriver。http://seleniumhq.wordpress.com/2013/08/28/the-road-to-selenium-3/

分是基于 JavaScript 的，因此经常会受到浏览器的安全限制。随着 Ajax（Asynchronous JavaScript and XML）的普及，越来越多的 Web 应用程序开始使用 JavaScript，这个问题也变得越发严重。

另一方面，从 2006 年前后开始，为了解决 JavaScript 限制的问题，Google 的 Simon Stewart 开始开发名为 WebDriver 的项目（图 7.6），通过调用 OS 的原生接口来操作浏览器。

图 7.6 使用 Selenium2 的构造



WebDriver 项目和 Selenium 项目能够互补，为用户提供更好的测试框架，基于这个原因，两者被合并在一起，并发布了 Selenium WebDriver①（以下称为 Selenium2）。Selenium2 发布后，原来的 Selenium RC 就被称为 Selenium1 了。

Selenium2 结合了 Selenium1 的“支持多种浏览器”“丰富的测试描述语言”的优点，以及 WebDriver 的“在 JavaScript 沙箱（sandbox）外

① http://www.seleniumhq.org/projects/webdriver/

部运行”“高速轻量且通用的浏览器仿真器”的优点[①]。

7.2.4 测试用例的制作和执行

下面我们就来看一下如何制作 Selenium 的测试用例。

Selemium IDE 的安装和运行

Selenium IDE 是 Firefox 的插件。用 Firefox 打开 `http://www.seleniumhq.org/download/`，点击最新版本的链接，即可进行安装。

启动 Selenium IDE，在浏览器上迁移画面、输入文字就可以直接将操作记录为命令，只需几分钟就可以制作好简单的测试用例。接着执行测试用例，就可以检验系统是否正常运行。

Selenium 的测试用例

简单地说一下 Selenium 的测试用例。Selenium 的测试用例可以用多种编程语言来制作，这里说明的是用 Selenium IDE 制作的 HTML 形式的测试用例。

Selenium 的测试用例由多个测试用例和测试套件（test suite）构成。

测试套件由多个测试用例组合而成。通过组合成为测试套件，多个测试用例就能按照指定的顺序执行（图 7.7）。

① 具体请参考 http://www.seleniumhq.org/docs/01_introducing_selenium.jsp#brief-history-of-the-selenium-project。

图 7.7　使用测试套件的结构示例



在 Selenium IDE 中建立多个测试用例后，点击“New Test Suite”就能建立测试套件。用编辑器打开建好的测试套件，就可以看到连接到各个测试用例的链接，如代码清单 7.1 和代码清单 7.2 所示。

代码清单 7.1　测试套件的例子（testsuite01.html）

```
<?xml version="1.0" encoding="UTF-8"?><!DOCTYPE html PUBLIC "-//W3C//DTD
XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.
dtd"><html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
lang="en"><head><meta content="text/html; charset=UTF-8" http-
equiv="content-type" /><title>Test Suite</title></head>
<body>
<table id="suiteTable" cellpadding="1" cellspacing="1" border="1"
class="selenium">
<tbody><tr>
    <td><b>login</b></td>
</tr>
<tr>
    <td><a href="login_successful.html">login_successful</a></td>
</tr>
<tr>
    <td><a href="login_failed.html">login_failed</a></td>
</tr>
</tbody>
</table>
</body>
</html>
```

代码清单 7.2　测试用例的例子（login_successful.html）

```
<?x邮件列表 version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.
w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head profile="http://selenium-ide.openqa.org/profiles/test-case">
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<link rel="selenium.base" href="http://example.com/" />
<title>login successful</title>
</head>
<body>
<table cellpadding="1" cellspacing="1" border="1">
<thead>
<tr><td rowspan="1" colspan="3">test</td></tr>
</thead><tbody>
<tr>
    <td>open</td>
    <td>/</td>
    <td></td>
</tr>
<tr>
    <td>waitForTextPresent</td>
    <td>ID、请输入密码</td>
    <td></td>
</tr>
<tr>
    <td>type</td>
    <td>id=loginId</td>
    <td>admin</td>
</tr>
<tr>
    <td>type</td>
    <td>id=loginPassword</td>
    <td>admin</td>
</tr>
<tr>
    <td>clickAndWait</td>
    <td>id=login</td>
    <td></td>
</tr>
<tr>
    <td>verifyTextPresent</td>
    <td>You are signed in to these accounts: </td>
    <td></td>
</tr>
</tbody></table>
</body>
</html>
```

●……… 什么是好的测试用例

虽说测试越快完成越好，但据说“用 Selenium 测试非常费时”。如果测试要耗费几个小时的话，那么频繁地执行测试就比较困难了，只能将多个修改合在一起进行测试，但这样测试失败时就难以分析原因了。并且若提交数天后才发现有 bug，开发的节奏也会受到影响，无法高效地进行。

制作测试用例、执行测试、确认结果，然后再制作测试用例、执行测试、确认结果……为了让开发人员能够像这样频繁地进行测试，测试再长也应该在 30 分钟内结束。为了缩短测试时间，在开始制作测试用例时就需要考虑下面这几点。

- **单独执行各个测试套件也必须能够通过测试**
- **1 个测试套件的执行时间不能超过所预期的全部测试的构建时间**

各个测试套件必须能够独立执行，依赖于其他测试套件的就说不上是好的测试套件。这是因为，如果测试套件之间相互依赖的话，就必须按照固定的顺序执行，这样一来就难以实现并行执行，测试的时间也会逐渐拉长。

相反，如果测试套件之间相互独立，就有可能将运行所有测试的时间缩短到和最长的测试套件的执行时间相同。因此在制作测试用例时要时刻意识到上述两点。

●……… 用 Selenium Server 来运行测试

Selenium IDE 在制作测试用例方面是非常强大的开发工具，但并不适用于定期、持续地执行测试用例。也就是说，开始测试时必须在运行浏览器的机器上点击 Selenium IDE 的开始测试按钮。随着测试用例的逐渐增加，这样的测试手段可以说将变得越来越不现实。

为了解决上述问题，下面将介绍使用 Selenium Server 来执行用 Selenium IDE 制作的测试用例的方法。

下载 Selenium Server 后①，通过下列命令运行 Selenium Server，就能够解析 HTML 格式的测试用例、启动作为测试对象的浏览器、执行测试并输出测试结果的报告。

```
C:\>java -jar selenium-server-standalone-<version-number>.jar -htmlSuite
"*firefox"
"http://example.com" "C:\path\to\testsuite01.html"
"C:\path\to\results.html"
```

命令的选项由 -htmlSuite<browser><startURL><suiteFile><resultFile>构成。可以通过选项 -h 来查看选项列表。

```
C:\>java -jar selenium-server-standalone-<version-number>.jar -h
```

来看一下上述命令的例子中 -htmlSuite选项的相关内容。

第 1 个参数“*firefox”表示进行测试的浏览器的种类。修改为“*iexplore”的话就可以使用 IE 进行测试。

第 2 个参数“http://example.com”是 Base URL。对系统的多个不同版本执行相同的测试用例的情况下使用该参数。测试用例的 open 命令所指定的 URL 的路径（相对路径）就是基于这个由 Base URL 指定的域名的，并作为绝对 URL 来执行。例如，在有开发团队 A 用的测试环境（http://a.example.com）和团队 B 用的测试环境（http://b.example.com）这样的情况下，可以使用这个参数。

第 3 个参数“C:\path\to\testsuite01.html”是测试套件文件的路径。请指定绝对路径。

第 4 个参数“C:\path\to\results.html”是测试结果报告的生成路径。同样要指定绝对路径。

这样通过使用 Selenium Server，就不必从 Selenium IDE 手动开始执行测试了，配合使用 Cron②或 Jenkins 就能够定期地执行测试，并且还能够在 Linux 或 Mac OS 等多个操作系统上测试多种浏览器。

至此，我们讲了使用 Selenium IDE 制作测试用例，利用 Selenium Server 执行测试的相关内容。这样就可以利用 Selenium 执行持续的集成测试了。

① 可以从 http://www.seleniumhq.org/download/ 下载。

② 自动执行任务的守护程序。

7.2.5　Selenium 的实际应用

●········ 测试页面是否有改动

根据系统或功能所要求的品质的级别，存在不允许预期以外的画面变更的情况。另外有的公司必须保存并提交画面的截图作为实施测试的证据。这时就可以使用 Selenium 的画面截图功能，实现打开画面后截图这种麻烦的作业的自动化。这里将介绍画面截图的方法，以及将获取的截图和以前的截图进行比较、测试的方法。

Selenium 可以通过“captureEntirePageScreenshot”或“captureEntirePageScreenshotAndWait”命令来截取当前浏览器的画面（图 7.8）。纵向较长的画面也能滚动后截取下来。在保存的文件名中输入“C:\path\to\img\capture1.png”这样的绝对路径来保存截图。不设定背景色的话会输出黑色的背景，所以可以预先设置好“background=#FFFFFF”等。

```
<tr>
    <td>captureEntirePageScreenshot</td>
    <td>C:\path\to\img\capture1.png</td>
    <td>background=#FFFFFF</td>
</tr>
```

接着说一下对保存的画面截图进行比较测试的方法。例如，将系统版本 1 时截取的画面和即将发布的版本 2 所截取的画面进行比较。对版本 1 和版本 2 执行相同的 Selenium 测试用例，各自输出同名的截图文件，分别存放在 C:\path\to\img\v1 和 C:\path\to\img\v2 这两个目录下。

图像的比较采用名为 ImageMagick[①] 的图像处理软件来进行。安装 ImageMagick[②] 后执行下列命令。可以利用本书提供的脚本[③] 来比较目录下的所有图像，并以 HTML 文件的形式输出结果。

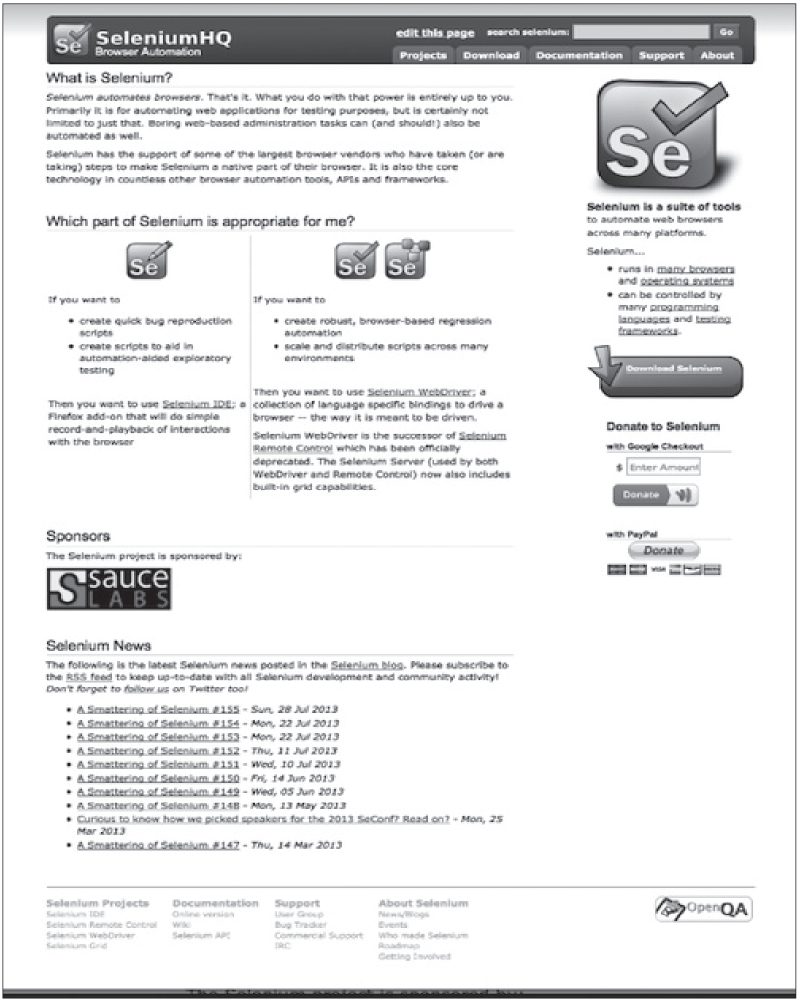
```
C:\>composite.exe -compose difference C:\path\to\img\v1\capture1.png
C:\path\to\img\v2\capture1.png C:\path\to\img\diff\capture1.png
```

① http://www.imagemagick.org/

② ImageMagick 的下载地址为 http://www.imagemagick.org/script/binary-releases.php。

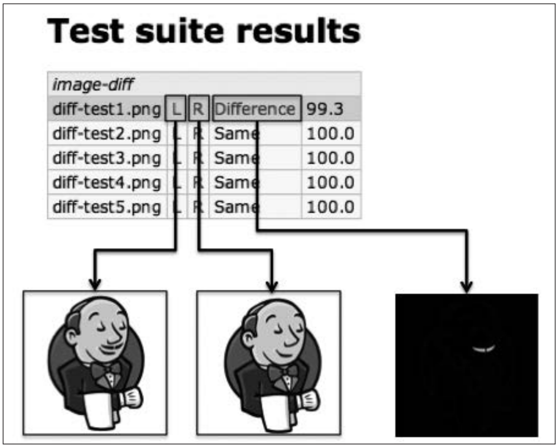
③ 打开 http://www.ituring.com.cn/book/1495，点击“随书下载”。

图 7.8 截取画面的例子



可以通过高光的图像来确认比较结果中存在差异的部分（图 7.9）。通过比较画面截图的方法，能够发现测试工程师无论如何测试都无法发现的问题，请一定要试一下。

图 7.9　附录脚本的执行结果文件



●········ 使 Selenium 测试稳定运行

Selenium 的自动测试并不是很稳定。经常听说系统明明正常运行，测试却失败了这样的事情。例如，点击链接，在画面加载完成前挂起下一步操作的 `clickAndWait` 命令执行时，点击链接迁移到下一个页面并试图点击页面内的按钮，但无法找到该按钮而造成测试失败。

上述情况下，因为 `clickAndWait` 命令执行成功，所以下一个页面应该已经加载完成了，那为什么无法点击本应正常显示的按钮而测试失败了呢？能够想到的原因有服务器内部的数据备份或执行的测试造成服务器的负载变高，对浏览器的响应速度变慢等。

但其实测试失败的本质问题在于服务器的响应速度慢，在页面内容还没有加载完毕的状态下 Selenium 就执行了下一条命令，从而导致测试失败。刚才的例子中，Selenium 虽然用 `clickAndWait` 命令等待画面加载，但没有等待该画面中接着要点击的按钮加载完毕就直接执行了下一条命令，所以造成了测试失败。

作为 Selenium 的创始人、公司的 CTO，Sauce Labs 在博客中这样叙述。

> "*Selenium tests often fail because they're too fast. Where a user might wait for a page to load for a few seconds and then click on a link, Selenium will interact with a page at the speed of code, before the page is ready.*"Selenium 的测试经常会因为速度过快而造成失败。人为操作的话会等待画面显示后再点击链接，Selenium 则是在画面完全显示之前以执行程序的速度实施测试的。
>
> —— How to Lose Races and Win at Selenium（http://sauceio.com/index.php/2011/04/how-to-lose-races-and-win-at-selenium/）

通常，人为操作浏览器时会等待按钮显示或页面加载完后再进行之后的操作，而 Selenium 则是直接继续之后的操作。

该如何解决这个问题呢？解决方法之一就是在使用`clickAndWait`命令等进行画面迁移之后，暂停（pause）固定数秒钟的时间，通过延迟执行下一条命令，多少能够所有改善。但这个方法为了解决偶尔发生的问题不得不在测试用例的各处插入`pause`命令，会造成测试执行时间变长，因此并不推荐。作为其他的解决方案，上述微博中还提到了下面的内容。

> "*The way to fix this is to have Selenium repeat its actions and assertions until they work. If you don't, Selenium races your browser.*"为了解决这个问题，Selenium 在得到正确的结果前必须重复动作或断言（assertions）处理，不这样的话 Selenium 就会和浏览器的画面显示速度持续竞争。
>
> —— How to Lose Races and Win at Selenium（http://sauceio.com/index.php/2011/04/how-to-lose-races-and-win-at-selenium/）

即命令执行失败时，重复执行该命令多次，直到正常运行为止。

但是用通过 Selenium IDE 做成的 HTML 形式的测试用例进行测试的情况下，是由 Selenium Server 来解析测试用例并操作浏览器的，因此不修改 Selenium Server 的代码就无法实现"重复数次执行该命令直到正常运行为止"。若要重复地实施处理，就需要通过编程语言（Java、Ruby、Python、Perl）来制作测试用例，自己添加重复处理。

下面介绍在用 Selenium IDE 制作测试用例的情况下，应该如何加入重复处理。按照下述步骤就可以修改从 Selenium IDE 导出的测试脚本中

的任意一条命令。

❶ 打开 Selenium IDE 的“Option”→“Option...”→“Formats”标签
❷ 选择作为例子所使用的格式（Java/JUnit4/RemoteControl 等），点击 source 按键，复制所显示的代码
❸ 点击 Add 按键，将步骤 2 复制的代码粘贴上去，对需要修改的地方进行修改
❹ 输入格式名称并保存
❺ 重启 Selenium IDE 和 Firefox
❻ 以步骤 4 做成的格式导出测试用例

将代码清单 7.3 修改为代码清单 7.4 那样，就可以反复执行确认画面中的文字内容是否显示的 verifyTextPresent命令。

代码清单 7.3　修改前的代码

```
function verifyTrue(expression) {
    return "verifyTrue(" + expression.toString() + ");";
}
```

代码清单 7.4　修改后的代码

```
function verifyTrue(expression) {
    return "for (int second = 0;; second++) {\n" +
        "\tif (second >= 60) fail(\"timeout\");\n" +
        "\ttry { " + (expression.setup ? expression.setup() + " " : "") +
        "if (assertTrue(" + expression.toString() + ")) break; } catch
(Exception e) {}\n" +
        "\tThread.sleep(1000);\n" +
        "}\n";
}
```

这样就能导出如代码清单 7.5 所示的测试脚本。在该测试脚本中，即使画面没有显示要确认的文字而造成 assertTrue 失败，经过一定时间后还会重新尝试确认。

代码清单 7.5　导出后的测试用例

```
for (int second = 0;; second++) {
    if (second >= 60) fail("timeout");
```

```
    try {
        if (assertTrue(selenium.isTextPresent("请输入ID、密码"))) break;
    } catch (Exception e) {}
    Thread.sleep(1000);
}
```

不要直接使用 Selenium IDE 制作的测试用例或以某种格式导出的测试脚本，根据测试脚本的制作方法的不同，确实能够减少由于页面加载时间所造成的测试失败。

7.3 Jenkins 和 Selenium 的协作

使用 Selenium 进行测试，测试用例的制作会大大加快。但是随着测试用例的增加，把握测试是否被正确执行、测试是否通过，也将变得非常麻烦。

对于这样的问题，第 5 章中所介绍的 Jenkins 这样的 CI 工具就非常有用。CI 工具可以实现测试的实施以及结果的品质状况可视化，并帮助团队间共享信息。

7.3.1　关联 Jenkins 和 Selenium 的步骤

下面将介绍由 Jenkins 执行 Selenium 测试用例的方法。

用 Selenium IDE 制作 Selenium 的测试用例并保存为 HTML 格式的话，通过使用 Jenkins 的"Seleniumhq Plugin"，从 Jenkins 执行 HTML 测试用例时的任务设置、测试结果报告的设置都将变得非常简单。

我们一起来看一下使用"Seleniumhq Plugin"的配置方法、测试执行以及测试结果的确认。这里以在运行有 Jenkins 的机器上执行 Selenium 测试为前提进行讲解。

❶ 在执行测试的机器上，从版本管理系统下载测试套件和测试用例[①]

❷在执行测试的机器上下载 Selenium Server[②]

❸ 从 Jenkins 的"系统管理"→"管理插件"安装"Seleniumhq Plugin"[③]

① 请参考 5.4 节中"下载代码"部分的内容。

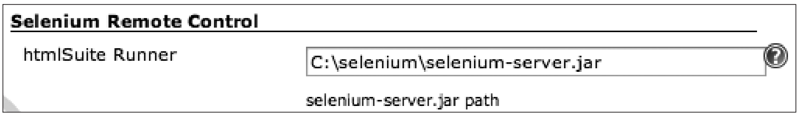
② Selenium Server 的下载用 Chef 来实现，请参考第 6 章。

③ https://wiki.jenkins-ci.org/display/JENKINS/Seleniumhq+Plugin

❹ 在 Jenkins 的“系统管理”→“系统设置”中设置“Selenium Remote Control”下的“htmlSuite Runner”（图 7.10）

设置为步骤 ❷ 中下载的 Selenium Server 的路径

图 7.10 Selenium Server 的设置画面



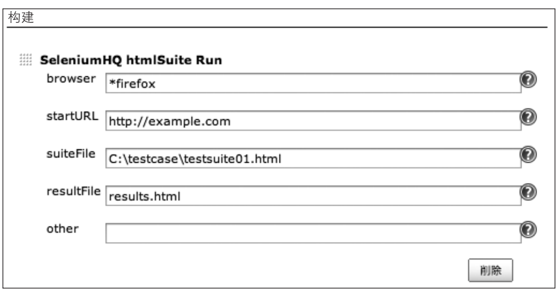
❺ 新建测试用的 Jenkins 任务，任务的“配置”如下

❺-1 在“构建”下的“增加构建步骤”中选择“SeleniumHQ htmlSuite Run”，按照表 7.1 进行设置（图 7.11）

表 7.1 SeleniumHQ htmlSuite Run 的设置

项目	说明
browser	设置执行测试的浏览器（Firefox 的话是 *firefox）
startURL	浏览器启动时打开的 URL
suiteFile	设置步骤 ❶ 中准备的测试套件
resultFile	设置测试结果的输出文件名（ex.results.html）

图 7.11 Selenium 测试任务的配置画面

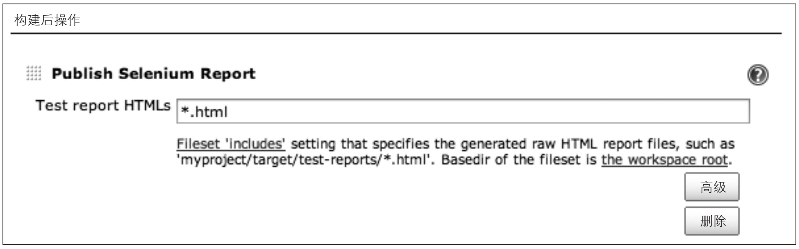


❺-2 选择“构建后操作”下“增加构建后操作步骤”中的“Publish Selenium Report”，按照表 7.2 进行设置（图 7.12）

表 7.2 构建后的处理

项目	说明
Test report HTMLs	指定报告的保存格式（ex.*.html）

图 7.12　Selenium 测试结果的设定画面

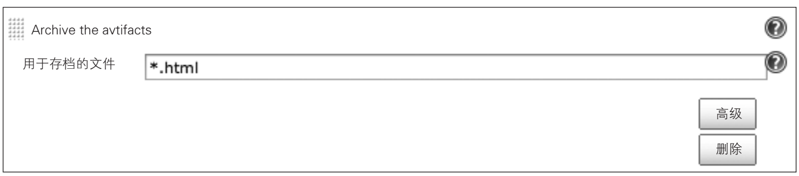


❺-3 选择“构建后操作”下“增加构建后操作步骤”中的“Archive the artifacts”，按照表 7.3 进行设置（图 7.13）

表 7.3　构建后的处理

项目	说明
用于存档的文件	指定保存文件的格式（ex.*.html）

图 7.13　Selenium 测试结果的保存画面



配置完成后，试着执行 Jenkins 的任务下的“立即构建”。配置正确的话会启动 Firefox 并开始测试，Selenium 测试最后会输出测试结果报告（图 7.14）。

当然也可以通过 Jenkins 确认测试的结果，试着从 Jenkins 任务画面左侧的 Build History 打开本次执行的链接。构建 #8 前面的○的颜色是蓝色的话表示构建成功，若是红色则表示构建失败（图 7.15）。

图 7.14 Selenium 测试执行过程中的画面

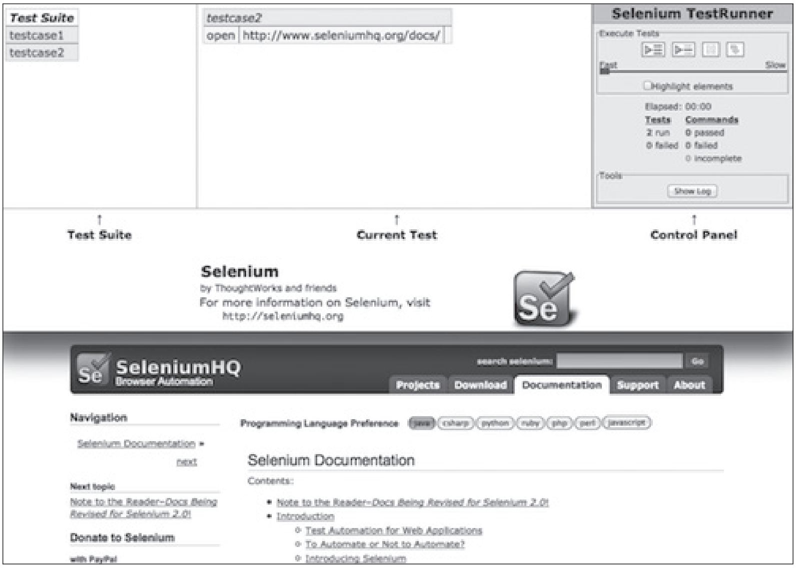
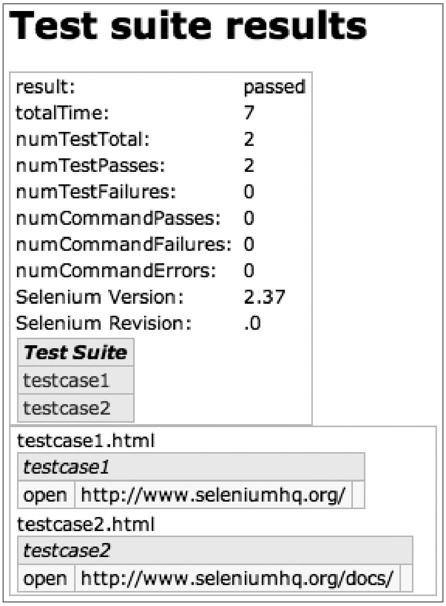


图 7.15 Jenkins 的构建结果画面



可以从“构建结果”来确认 Selenium 的执行结果（图 7.16）。测试失败时可以从这个 Selenium 的执行结果或“Console Output”查看失败的原因。

图 7.16　Jenkins 的 Selenium 测试结果画面



7.4 Selenium 测试的高速化

之前已经多次提到，Selenium 的测试中经常会出问题的就是测试的速度。Selenium 的测试是从使用浏览器的系统用户的观点出发的集成测试，加上 Selenium 自身的速度并不快，因此和单元测试相比的确是非常耗费时间的测试。

另外，在用 Selenium IDE 制作测试用例的情况下，因为非常简单，所以往往容易不加思索地对各处进行测试，结果就导致执行所有的测试需要耗费大量的时间。

下面是笔者所在公司的 Selenium 测试套件数量的演变图（图 7.17）和所有测试套件执行时间的演变图（图 7.18）。随着自动测试的不断增加，执行所有测试甚至要花费数日之久。

图 7.17　Selenium 测试套件数量的演变

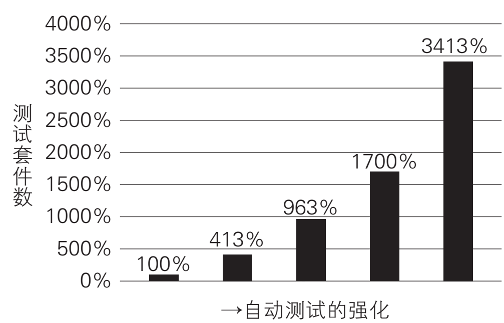
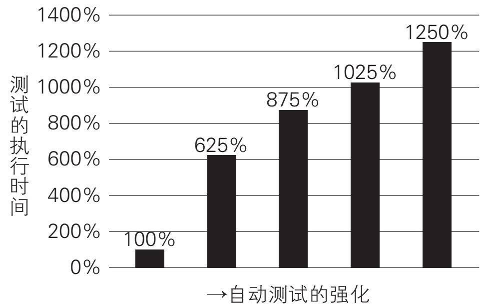


图 7.18　Selenium 测试执行时间的演变



作为上述问题的有效解决方案，本节将介绍并行执行测试的方法。

7.4.1　利用 Jenkins 的分布式构建实现测试的并行执行

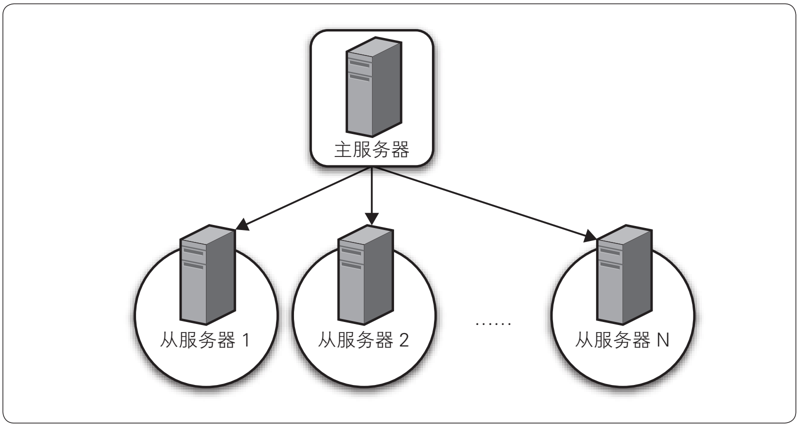
首先，正如之前在“好的测试用例”一节中所讲的那样，开始并行执行测试时，各个测试套件必须不相互依赖，能够单独通过测试。在此基础上再利用 Jenkins，即可构建并行执行的机制。

Jenkins 提供了“分布式构建”这一功能强大的机制来支持并行测试。这里对基于上述机制的并行执行进行讲解。

●……… Jenkins 的分布式构建的构成

利用 Jenkins 的“分布式构建”机制能够组建从数十台到数百台机器的集群（图 7.19）。

图 7.19　Jenkins 的主（master）和从（slave）结构



各台机器根据功能的不同分为 master 和 slave 两种。安装有 Jenkins 的机器作为 master 负责各种配置信息的管理、构建时处理流程的控制，以及调度各 slave 实施处理等。

slave 从 master 接收指令，从版本管理系统上取得、更新代码或测试用例并执行测试。

●……… 分布式构建的配置

接着就让我们试着实际构建一下 Jenkins 的 master 和 slave 架构。这里以 Windows 平台为例讲解 slave 的架构。

❶ 配置 slave 机器的 Jenkins

从 Jenkins 的“系统管理”→“管理节点”→“新建节点”开始新建 slave 节点（图 7.20）。slave 的种类选择“Dumb Slave”。其他项目的设置如表 7.4 所示。

图 7.20 配置 slave 机器的 Jenkins

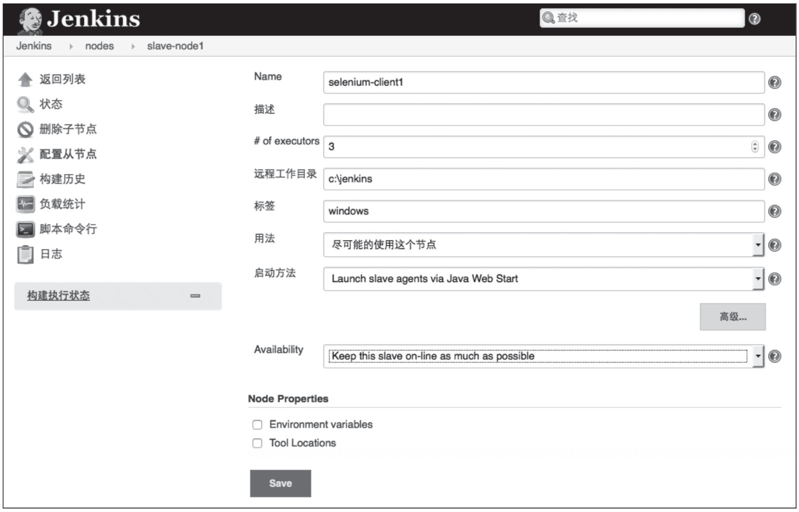


表 7.4 slave 的设置

项目	说明
Name	设置 slave 节点的名称。以 slave 的计算机名来命名简洁易懂
描述	设置节点的描述
of executors	设置 slave 机器上能够同时构建的任务数量。可以先设置为 3 或 5，确认执行测试时的负载后再进行调整
远程工作目录	slave 机器上 Jenkins 的根目录。可以设置为 c:\jenkins 等
标签	可以为各 slave 机器标注标签。输入例如“windows”这样提示 slave 特征的标签
用法	指定构建的调度方法。设置为默认的“尽可能使用这个节点”
启动方法	选择“Launch slave agents via Java Web Start”
Availability	指定 slave 的可用性。设置为默认的“Keep this slave on-line as much as possible”

❷ master 和 slave 的连接

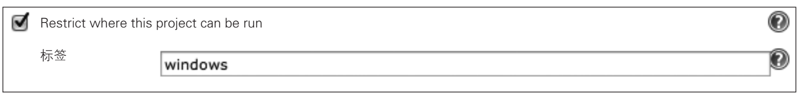
从新添加的 Windows 节点的浏览器打开 Jenkins，点击步骤 ❶ 中建立的 slave 画面上的“Launch”键，就能下载 JNLP 文件。运行该 JNLP 文件，就会安装作为 slave 节点所需要的文件，并连接 master。Windows 会因为 Windows update 而频繁地重启，因此要进行如下设置。

- 将下载的 JNLP 文件存放到启动目录下
- 设置为自动登录

❸ 配置测试任务在哪个 slave 上执行

在各任务配置的“Restrict where this project can be run”（图 7.21）中指定让哪个 slave 来执行构建处理。这里可以填入步骤 ❶ 中设置的 slave 节点名、slave 的标签，或者标签和逻辑运算符（&& 或 || 等）的组合，由符合这里设定的限制条件的 slave 节点之一来执行构建处理。可以直接在“Restrict where this project can be run”设置 slave 节点名，但这并不是聪明的方法，因为随着测试任务的增加，特定 slave 的负担加重，即使构建了分布式的环境，但因为限制了特定的 slave，也会造成无法顺利均衡负载。因此理想的方法是使用 slave 的标签。

图 7.21 配置测试任务在哪个 slave 上执行



❹ 执行构建

点击 Jenkins 任务的“立即构建”就会启动 slave 机器执行测试。这样分布式构建环境的构筑就完成了。

7.4.2 Selenium 测试并行化中的难点

至此，我们讲解了为了尽快完成测试，实现 Selenium 测试的高速化，测试用例应该符合哪些条件，以及并行执行多个测试用例的方法。

这里再介绍一下笔者所在公司并行执行测试时所发生的问题。虽然在多数系统上该问题未必会发生，但可以为解决推进 Selenium 测试并行化过程中的若干问题提供一些启示。

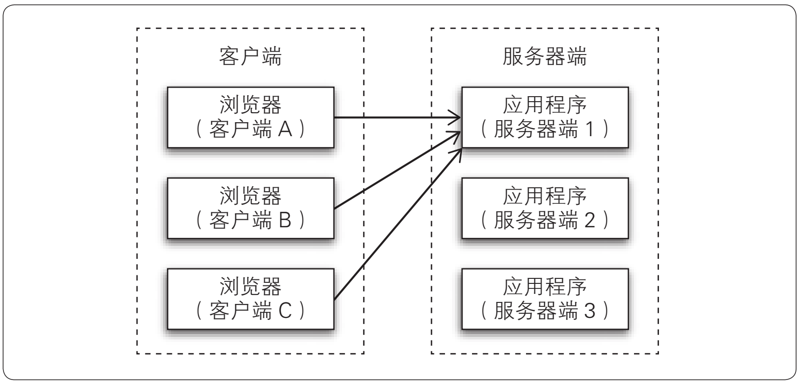
在之前的“使 Selenium 测试稳定运行”这一小节中，我们介绍了 Selenium 会无视当前页面内容是否加载完成，而直接继续下面的测试，因而造成了系统虽然正常运行但测试依然失败的情况。造成这种现象的

原因可能是服务器正在备份或解析日志等而造成服务器的负载变高。

同样，并行执行测试也会引发新的问题，那就是并行执行会启动多个浏览器，这些浏览器的请求会集中发往特定的服务器，这样该服务器的响应就会变得缓慢。

根据上述方法并行执行 Selenium 测试的情况下，Jenkins 任务中指定的 slave 或标签就是会启动浏览器的机器。上述方法可以实现运行浏览器的机器的负载均衡，但无法实现从浏览器接收请求的服务器的负载均衡（图 7.22）。

图 7.22　Selenium 测试并行执行时的请求不均衡



这个问题虽不会造成测试失败频繁发生，但在运行系统的服务器配置较低或系统的处理能力较差等情况下就很容易造成这样的问题。

对于上述问题，我们采取的对策是将浏览器客户端和运行系统的服务器进行 1∶1 配对（图 7.23），包括应用程序用到的服务器端的缓存服务器以及数据库服务器等，都和浏览器客户端组成 1 个配对，通过增加浏览器客户端和服务器端的配对形式来扩展测试的规模。

要使浏览器客户端和服务器进行 1∶1 的配对，就需要替换各个机器上的 hosts 文件，使其向特定的机器发送请求（图 7.24）。例如，替换浏览器客户端机器上的 hosts 文件，使其向对应的应用程序服务器发送请求。

这样特定机器上运行的浏览器的请求就会发往特定的服务器进行处理。服务器端也一样，固定服务器上运行的应用程序会使用其对应的数据库服务器。

图 7.23　1 : 1 的浏览器客户端和服务器的结构

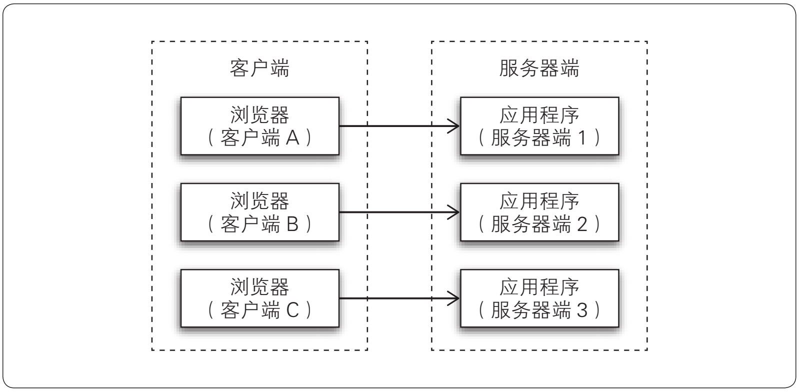
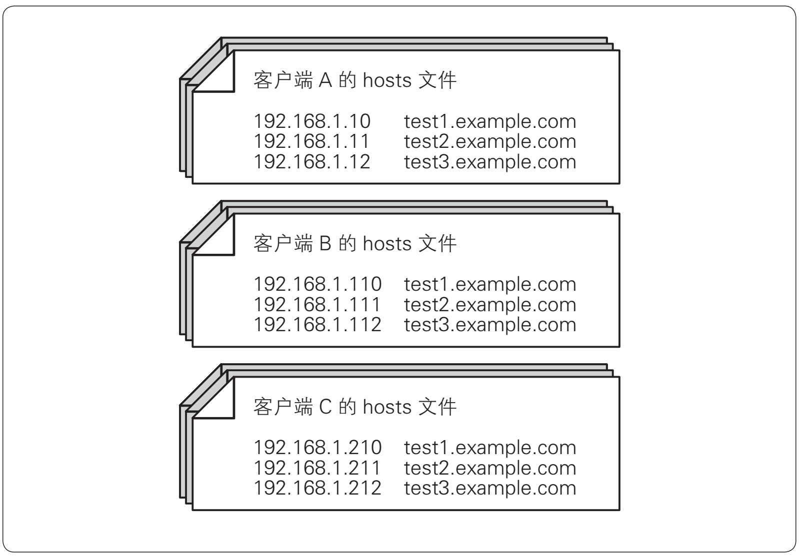


图 7.24　hosts 文件的修改



如上所述，使用 Selenium 进行的是集成测试，所以和单元测试的并行执行不同，浏览器端的机器、服务器端的机器或数据库服务器等，其中任意一项的高负载都会造成测试不稳定。根据具体的原因处理方法也有所不同。在推进测试的并行执行过程中，机器的负载是需要注意的问题之一。

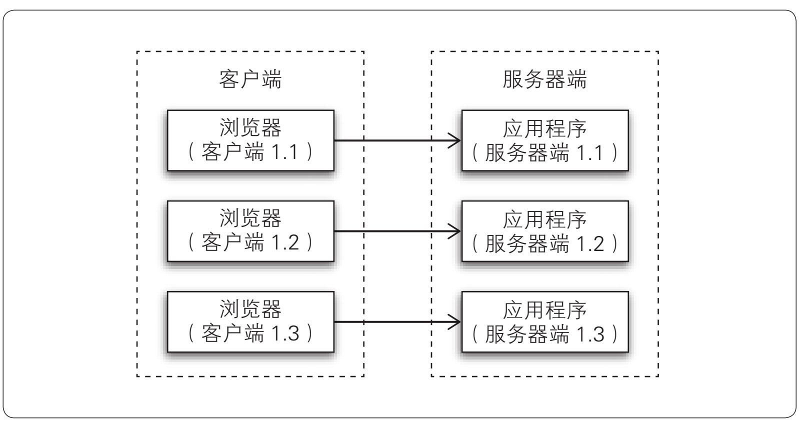
7.5 多个应用程序版本的测试

第 3 章中介绍了灵活运用版本管理系统是顺利进行团队开发所必不可少的要素。当然不仅是代码，测试用例也应该进行版本管理。这样就可以对系统的多个版本进行测试，其优点有如下这些。

- **在发布后的版本中发现 bug，需要紧急发布的情况下，如果能够实施自动化测试，就能够安心地发布**
- **能够和紧急发布的测试并行进行下一次发布的开发工作**
- **每个开发团队在不同的分支上开发的情况下，可以在该分支上执行其对应的测试用例，在向主代码库提交之前实施自动测试**

这里介绍一下对多个版本进行 Selenium 测试的方法（图 7.25）。首先，用 Jenkins 的“Parameterized Trigger pluguin”来指定对哪个版本执行测试。假设这里设置的变量名字为 `APP_VERSION`，那么在 Jenkins 的任务配置中，用名为 `${APP_VERSION}` 的变量就能在构建时取得指定的版本。

图 7.25 对多个版本进行 Selenium 测试



7.5.1　应用的部署

通常，对应用程序的多个版本进行单元测试时，只需要将代码变更为特定的版本就可以进行测试了。而 Selenium 测试则需要从部署应用程序的特定版本开始着手。因此，测试的整体流程可分为“部署应用程序”和“用 Selenium 测试”。可以使用 Jenkins 的构建流水线或任务关联等来构筑这样的流程。

7.5.2　从版本管理系统下载测试用例

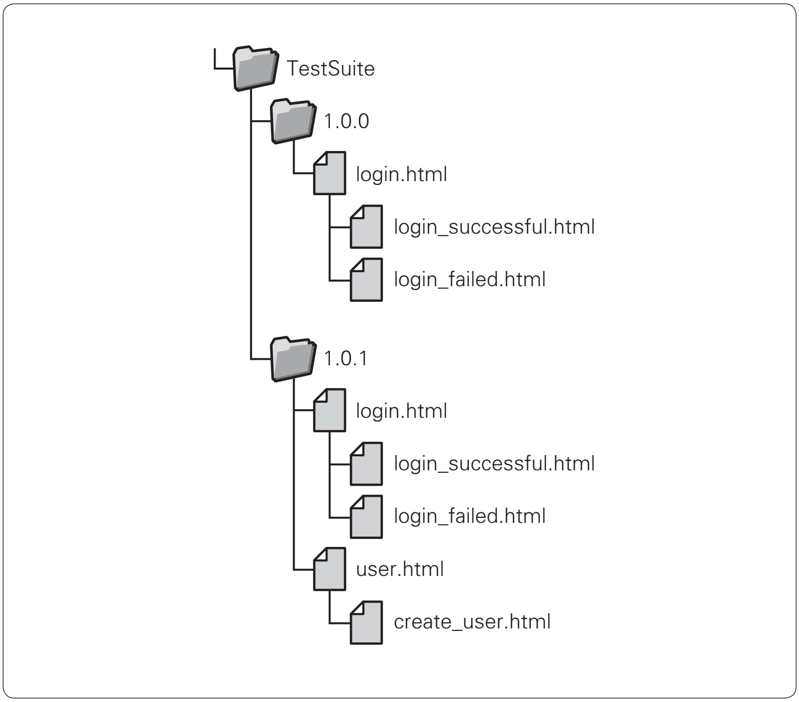
接着从版本管理系统上下载测试用例。当然前提是测试用例和代码一样都进行了版本管理。

在 Jenkins 的测试任务的工作目录下，以各版本为目录名保存测试用例（图 7.26）。使用 Git 管理代码的情况下，可以将 Git 插件的“高级”下的“Local subdirectory for repo（optional）”设置为“`TestSuite/${APP_VERSION}`”。如果忘记进行上述配置，各版本的测试用例就会被直接下载到测试任务的工作目录下，和其他版本的测试用例混在一起，因此需要特别注意。

7.5.3　用 Selenium 测试

正如 7.3 节中讲解的那样，只需在“seleniumhq Plugin”的 suiteFile 中设置刚才下载的测试套件，即可执行测试。可以将 suiteFile 设置为“`${WORKSPACE}/TestSuite/${APP_VERSION}/login.html`”，这样就可以对多个版本进行测试了。

图 7.26 保存各版本的测试用例



7.6 本章总结

如本章所述，自动化的回归测试能够使发生退化 bug 的风险最小化，同时又是能快速、持续地进行功能添加和改善的有效手段。除此之外，本章还介绍了作为其工具的浏览器驱动测试工具 Selenium，以及为了高效地实施 Selenium 测试而进行的 Selenium 和 Jenkins 之间的关联。

回归测试不仅是开发人员实施的单元测试，还包括从用户视角进行的集成测试、用户验收测试。在持续的系统版本更新时，这些是非常强有力的测试手段。

但是要补充一下，回归测试的自动化未必会使测试代码减少。使用 Selenium 的测试自动化有时会需要大量的维护工作。也有比较适合进行手动测试的情况。因此在制作自动化测试时，可以先思考一下正要制作的测试今后会被执行几次，今后需要怎样的维护等。

参考文献・网址

……全部

Andrew Hunt、David Thomas（著），马维达（译），程序员修炼之道：从小工到专家，电子工业出版社，2011

WEB DB PRESS 編集部編集，開発ツール徹底攻略，技術評論社，2013

菅野裕、今田忠博、近藤正裕、杉本琢磨，Trac 入門，技術評論社，2013

Michael C. Feathers（著），候伯薇（译），修改代码的艺术，机械工业出版社，2014

Ken Schwaber、Jeff Sutherland（著），王军（译），30 天软件开发：告别瀑布拥抱敏捷，人民邮电出版社，2014

Brian W. Fitzpatrick 、Ben Collins-Sussman，Team Geek: A Software Developer's Guide to Programming Well with Others，O'Reilly Media，2012

Avram Joel Spolsky，Joel on Software，APress，2004

G.Pascal Zachary（著），张银奎（译），观止 -- 微软创建 NT 和未来的夺命狂奔，机械工业出版社，2009

……Git、版本管理系统

大塚弘記，Git 実践入門——Pull Request による開発の変革，技術評論社，2014

濱野純，入門 Git，秀和システム，2009

Travis Swicegood，Pragmatic Version Control Using Git，The Pragmatic Programmers，2009

Pro Git（http://git.oschina.net/progit/）

猴子都能懂的 GIT 入门（http://backlogtool.com/git-guide/cn/）

The version timeline（http://codicesoftware.blogspot.com/2010/11/version-control-timeline.html）

Astonishments,ten,in the history of version control（http://www.flourish.org/blog/?p=397）

分散バージョン管理で間違いないって、ベイビー（http://local.joelonsoftware.com/wiki/ 分散バージョン管理で間違いないって、ベイビー）

A successful Git branching model（http://nvie.com/posts/a-successful-git-branching-model/）

GitHub Flow（http://scottchacon.com/2011/08/31/github-flow.html）

……Scrum、敏捷开发

Mike Cohn（著），宋锐（译），敏捷估计与规划，清华大学出版社，2007

西村直人、永瀬美穂、吉羽龍太郎，SCRUM BOOTCAMP THE BOOK，翔泳社，2013

平鍋健児、野中郁次郎，アジャイル開発とスクラム，翔泳社，2013
長瀬嘉秀（監修）、高畠勇人 、渡辺裕 、株式会社テクノロジックアート（著），アジャイル開発マネジメント クイックガイド ，技術評論社，2013
前川直也、西河誠，わかりやすいアジャイル開発の教科書，SB クリエイティブ，2013

……build・测试

Janet Gregory、Lisa Crispin（著），孙伟峰、崔康（译），敏捷软件测试：测试人员与敏捷团队的实践指南，清华大学出版社，2010

渡辺修司，JUnit 実践入門，技術評論社，20

Srirangan，Apache Maven 3 Cookbook，Packt Publishing，2011

野瀬直樹、横田健彦，Apache Maven 2.0 入門 Java· オープンソース · ビルドツール，技術評論社，2006
川口耕介，詳解 Jenkins，WEB DB PRESS Vol.67，技術評論社，2012
Selenium-Web Browser Automation（http://www.seleniumhq.org/）

読書メモ『実践アジャイルテスト』（http://somat.hatenablog.com/entry/20100919/1284922305）

Selenium 何とかっていうツールがやたら色々あるのはどういうわけなのか（http://blog.trident-qa.com/2013/05/so-many-seleniums）

日本 Selenium ユーザーコミュニティ——Selenium IDE からソース出力する際のフォーマットの変更について——（https://groups.google.com/forum/#!topic/seleniumjp/1PoYX-Qmu3k）

……持续集成

Paul M. Duvall 、Steve Matyas、Andrew Glover（著），王海鹏（译），持续集成：软件质量改进和风险降低之道，电子工业出版社，2012 年

John Ferguson Smart，Jenkins: The Definitive Guide，O’Reilly Media, Inc, USA，2011

佐藤 聖規（著、監修）和田貴久、河村雅人、米沢弘樹、山岸啓（著），Jenkins 実践入門　~ビルド・テスト・デプロイを自動化する技術，技術評論社，2011

……持续交付

David Farley、Jez Humble（著），乔梁（译），持续交付：发布可靠软件的系统方法，人民邮电出版社，2011
伊藤直也，Chef Solo——Infrastrucuere as Code，2013
新原雅司，入門ガイド，技術評論社，2013
Open Source Provisioning Toolchain（http://www.slideshare.net/dev2ops/velocity-online-provisioningtoolchainkey）

版权声明

员工："老板，我要加工资！"
老板："为什么？"
员工："因为我长得帅！"
老板："……"
员工："因为我跟你10年了，没有功劳也有苦劳吧！"
老板："好吧，加5%差不多了。"
员工："这个项目交给我，我有办法只需要一半的人手就能完成！"
老板："真的？好！工资翻倍！"

——摘自本书译者序

- ☹ 重要的邮件太多而无从下手
- ☹ 没有能用于验证的环境
- ☹ 覆盖了其他组员修正的代码
- ☹ 无法自信地进行代码重构
- ☹ 不知道bug的修正日期，也不能追踪退化
- ……

那么，你可能需要这本书！

★ 系统讲解团队开发所必需的工具和方法

★ 详细介绍各个工具的特性及使用要点，并进行比较

★ 自动化意识贯穿全书，真正实现高效开发

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机/软件工程与软件方法学

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-29594-1

ISBN 978-7-115-29594-1
定价：49.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks